Martim Azevedo do Nascimento

Um algoritmo para o cálculo de cobertura de estados

Dissertação submetida ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina para a obtenção do Grau de Mestre em Ciência da Computação.
Orientadora: Prof.ª Dra. Patrícia Vilain

Florianópolis / SC
2015

Ficha de identificação da obra elaborada pelo autor
através do Programa de Geração Automática da Biblioteca Universitária
da UFSC.

Nascimento, Martim Azevedo do
Um algoritmo para o cálculo de cobertura de estados /
Martim Azevedo do Nascimento ; orientadora, Patrícia Vilain
- Florianópolis, SC, 2015.
104 p.

Dissertação (mestrado) – Universidade Federal de Santa
Catarina, Centro Tecnológico. Programa de Pós-Graduação em
Ciência da Computação.

Inclui referências

1. Ciência da Computação. 2. Adequação de testes de software.
3. Cobertura de estados. 4. Cobertura de código. I. Vilain, Patrícia.
II. Universidade Federal de Santa Catarina. Programa de Pós-
Graduação em Ciência da Computação. III. Título.

Martim Azevedo do Nascimento

UM ALGORITMO PARA O CÁLCULO DE COBERTURA DE ESTADOS

Esta Dissertação foi julgada adequada para obtenção do Título de "Mestre", e aprovada em sua forma final pelo Programa Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina.

Florianópolis, 03 de março de 2015.

_____
Prof. Ronaldo dos Santos Mello, Dr.
Coordenador do Curso

Banca Examinadora:

_____
Prof.ª Patrícia Vilain, Dr.ª
Orientadora
Universidade Federal de Santa Catarina

_____
Prof. Raul Sidnei Wazlavick, Dr.
Universidade Federal de Santa Catarina

_____
Prof. Ricardo Pereira da Silva, Dr.
Universidade Federal de Santa Catarina

_____
Prof ª. Juliana Herbert, Dr. ª
Universidad de la Republica Uruguay

Este trabalho é dedicado à minha filha Sara.

AGRADECIMENTOS

Agradeço a Deus por tantas coisas boas que tem me dado.
Agradeço meus pais e minha irmã que sempre estiveram do meu lado nos momentos difíceis e nos momentos felizes. Esse é um momento feliz portanto estão comigo.
Agradeço minha filha Sara pela inspiração e pela motivação nos dias mais cansativos. É para que tenha orgulho de mim, o motivo deste esforço. Agradeço pela compreensão nos momentos de ausência.
Agradeço minha companheira Gabi pela serenidade e paciência que tanto me ajudam a passar por estes caminhos árduos mas recompensadores. Obrigado pelos conselhos valiosos e pela dedicação na construção de uma família.
Agradeço minha orientadora Patrícia Vilain por ter acreditado no meu trabalho e pelo empenho e dedicação na orientação.

RESUMO

Cobertura de estados é um critério de adequação de testes de software que mede a quantidade de modificações de estados feitas durante a execução dos testes que foram verificadas através de asserções. O presente trabalho propõe um algoritmo para o cálculo de cobertura de estados baseado em construções comuns a linguagens orientadas a objetos, como atribuições, retorno de métodos e chamadas de funções. O algoritmo identifica modificações de estados cobertas por asserções através de um novo cálculo de influências de atributos em métodos de uma classe baseado na extração de dependências entre identificadores existentes no código. São apresentados ainda uma extensão à definição de estado modificado levando em consideração a distinção entre atributos simples e compostos (estruturas de dados) e uma implementação do algoritmo através da instrumentação de bytecode Java. Experimentos feitos em sete projetos de código aberto mostraram a utilidade do algoritmo na identificação de atributos não verificados por asserções mas executados pelo teste. Ao inserir erros propositais nestes atributos os testes não falharam. Em uma situação real, estes erros estariam imperceptíveis pelos desenvolvedores. Os experimentos ainda mostraram que a execução do algoritmo adicionou um *overhead* de 20% a 33% no tempo de execução dos testes unitários, um valor abaixo dos trabalhos existentes.

Palavras-chave: Adequação de testes de software. Cobertura de estados. Cobertura de código.

ABSTRACT

State coverage is a test adequacy criterion that measures the quantity of state modifications made by a test execution that were verified by assertions. This work proposes an algorithm for state coverage based on common constructions of object-oriented languages, such as assignments, method returns and function calls. The algorithm identifies influences of an instance attribute on assertions applying a new method to compute influences of attributes on methods. This work also presents an extension to the modified state definition by applying a distinction between simple and compound types (data-structures) and an implementation of the algorithm based on instrumentation of Java bytecode. Experiments made on seven open source projects showed the validity of the algorithm in identifying attributes not verified by assertions that were executed by tests. Bugs inserted in these attributes were not captured by the tests. The results also showed that the algorithm adds an overhead of 20% and 33% at the test execution. These values are below of existent works.

Keywords: Software test adequacy. State coverage. Code Coverage.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

ODS – *Output-Defining Statements*
MC/DC – *Modified Condition / Decision Coverage*
JSON – *JavaScript Object Notation*
HTML – *HyperText Markup Language*

Sumário

1 INTRODUÇÃO

Proposta inicialmente por Koster et al (2007), cobertura de estados tem como objetivo encontrar testes sem asserções ou asserções fracas. Ao contrário dos critérios de cobertura tradicionais baseados na execução de linhas de código (cobertura de caminhos, por exemplo), a cobertura de estados tem como interesse a quantidade de asserções feitas pelos testes que verificam os estados modificados pelo mesmo. Baseia-se na intuição de que um teste pode executar todos os caminhos de execução possíveis no programa mas ainda assim não possuir nenhua verificação do estado resultante. Sem uma asserção nas saídas do programa não é possível garantir com precisão a não existência de erros no mesmo.

Para demonstrar tal problema considere o código apresentado na figura 1.

Figura 1: Código demonstrando um teste sem asserção

```
class Color {
   private String name;
   private int rgbValue;

   public Color(String name, int rgbValue) {
      this.name = name;
      this.rgbValue = rgbValue;
   }

   public String getName() {
      return name;
   }
 }

@Test
public void testWithoutAssert() {
   Color color = new Color("green"
                           , 0x00ff00);
   System.out.println(color.getName());
}
```

Na execução do teste *testWithoutAssert* (Figura 1) todos os caminhos do programa são executados, atingindo uma adequação de 100% em cobertura de linhas de código. No entanto, a inserção de um

erro na lógica do programa não irá ser capturada pelo teste pois não há nenhuma verificação no resultado do programa. É esse tipo de problema que a cobertura de estados tenta evidenciar.

1.1 CONTEXTUALIZAÇÃO DO PROBLEMA

De maneira geral define-se a cobertura de estados para um teste como sendo a razão entre a quantidade de modificações de estado verificadas por asserções pela quantidade total de modificações de estado.

$$Cobertura\ de\ estados = \frac{\text{Quantidade de modificações de estado verificadas}}{\text{Quantidade total de modificações de estado}}$$

Considera-se como estado de um programa orientado a objetos os valores dos atributos dos objetos de suas classes.

Um atributo é considerado modificado se existir alguma instrução na execução do teste que altere o seu conteúdo.

Um atributo é considerado verificado se influencia alguma asserção dentro do teste, ou seja, se o seu valor é o próprio valor sendo verificado na asserção ou se participa da computação do valor sendo verificado. No código da Figura 2 o teste *testWithAssert* exercita a classe *Color* mostrada anteriormente. No entanto, agora há uma asserção no retorno do método *getName*. Como o valor do atributo *name* participa da computação do resultado de *getName* então o mesmo é considerado coberto.

Figura 2: Asserção cobrindo um atributo

```
@Test
public void testWithAssert() {
  Color color = new Color("green", 0x00ff00);
  assertTrue(color.getName() == "green");
}
```

Para ser considerado coberto o atributo deve influenciar o resultado da asserção. Não é suficiente que o atributo tenha seu valor utilizado na execução de um método sem que tenha influência no seu resultado. Se o método *getName* fizesse uma leitura do valor do atributo *rgbValue* sem

causar influência em seu retorno este atributo não poderia ser considerado coberto. A Figura 3 mostra esta situação.

Figura 3: Método utilizando um atributo sem que participe do seu resultado

```
public String getName() {
   // o atributo rgbValue é utilizado mas
   // não influencia o resultado do método
   System.out.println(
           "Rgb value : " +
                       this.rbgValue);
   return name;
}
```

Saber se um atributo tem influência no resultado de uma asserção é um problema fundamental para o cálculo de cobertura de estados. Os trabalhos existentes na literatura de cobertura de estados ou não apresentam um algoritmo eficiente para resolver este problema (KOSTER et al, 2007; KOSTER, 2008) ou não apresentam uma solução para isto (VANOVERBERGHE et al, 2012).

Cobertura de estados é uma métrica promissora que poderá auxiliar na identificação de testes fracos. Tem o propósito de complementar a cobertura de código e de maneira nenhuma pode substituí-la, pois ambas medem aspectos diferentes do programa. Enquanto a cobertura de código tem como medida de adequação as construções do código executadas pelos testes, a cobertura de estados tem foco nas asserções dos testes. Seu uso tem como objetivo fornecer pistas para o testador sobre asserções inexistentes para as mudanças de estado do programa.

No entanto, apesar de promissora, não se conhece nenhum algoritmo para cobertura de estados completo com implementação trivial, eficiente e que utilize construções comuns a linguagens de programação disponível para uso e experimentações.

O presente trabalho propõe um algoritmo com essas características, apresentando um novo cálculo de influências entre atributos e métodos e demonstrando o seu uso em projetos de código aberto.

O trabalho apresenta ainda uma extensão à definição de estado modificado, abrangendo a distinção entre atributos de tipo simples e de tipo composto (estruturas de dados).

1.2 OBJETIVOS

O trabalho apresenta os seguintes objetivos:

1.2.1 Objetivo geral

Apresentar um novo algoritmo para o cálculo de cobertura de estados, com ênfase em construções comuns a linguagens de programação.

1.2.2 Objetivos específicos

1. Propor um novo algoritmo para a verificação de influências entre atributos e métodos para ser utilizado no cálculo de cobertura de estados.
2. Apresentar uma extensão à definição de estado modificado, levando em consideração a distinção entre atributos de tipo simples e de tipo composto.
3. Desenvolver uma implementação do algoritmo em uma linguagem de programação de propósito geral.
4. Realizar experimentos utilizando esta implementação em projetos de código aberto.

1.4 ESTRUTURA DO TRABALHO

O trabalho está organizado da seguinte forma. No Capítulo 2 é apresentada a fundamentação teórica sobre adequação de testes de software explicando os conceitos de cobertura de código e cobertura de estados que serão utilizados no texto. No Capítulo 3 é apresentada uma revisão bibliográfica da área. No Capítulo 4 é apresentado o algoritmo proposto no trabalho. Para isso, primeiro são feitas explanações sobre o cálculo de cobertura de estados, apresentando os problemas existentes a serem resolvidos. Na sequência apresenta-se a definição do algoritmo para a verificação de influências de atributos em métodos bem como a definição do algoritmo de cobertura de estados propriamente dito. No Capítulo 5 é apresentada a implementação do algoritmo de cobertura de estados proposto utilizando a linguagem Java. No Capítulo 6 são apresentados os resultados do experimento de uso do algoritmo em projetos de código aberto. No Capítulo 7 são expostas as diferenças entre este trabalho e os existentes. Por fim, no Capítulo 8 são apresentadas as conclusões e possibilidades de trabalhos futuros.

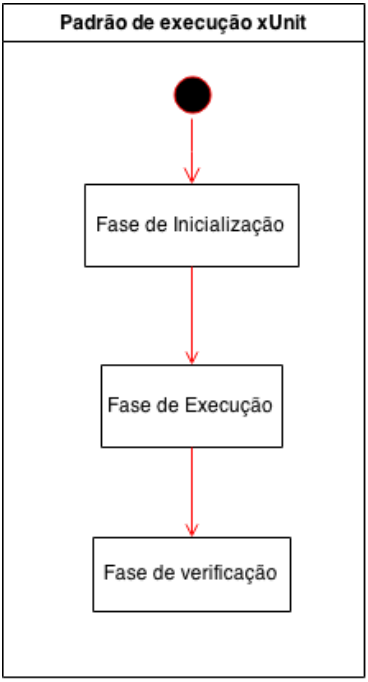
2 FUNDAMENTAÇÃO TEÓRICA

2.1 FUNDAMENTOS DE TESTE DE SOFTWARE

Teste de software é uma prática importante durante o desenvolvimento de sistemas. Apesar de não provar que um programa está livre de erros (DIJKSTRA, 1972) é uma forma eficiente de minimizar a existência destes (RAPPS, 1985).

Considera-se para este trabalho a estrutura de testes proposta por *frameworks* no padrão *xUnit*. (MESZAROS, 2007). Neste padrão um teste é formado por três fases: fase de inicialização, fase de execução e fase de verificação. Na fase de inicialização são feitas as inicializações nas classes de produção (as classes sendo testadas). Na fase de execução o teste executa os métodos destas classes. Por último, na fase de verificação são feitas asserções em cima do estado das classes de produção após a execução do teste (Figura 4).

Figura 4: Padrão de execução de frameworks xUnit



2.1.1 Asserções

Asserções são predicados (funções que retornam verdadeiro ou falso) que validam as saídas de um programa, interrompendo a execução caso estas saídas estejam incorretas de acordo com a sua especificação. Podem ser encontradas junto às instruções das classes de produção (*inline*) ou nos testes destas classes. Este trabalho assume que as asserções são feitas nos testes da classe durante a fase de verificação. *Frameworks* no padrão *xUnit* disponibilizam bibliotecas de asserções pré-definidas. A Figura 5 mostra exemplos de asserções do framework JUnit (2015).

Figura 5: Exemplo de asserções no framework JUnit

```
assertEquals(0, new ArrayList().size());
assertTrue(new ArrayList().isEmpty());
assertFalse(1 + 1 == 3);
```

2.1.1.1 Expressões de uma asserção

Uma asserção é formada por uma ou mais expressões. Considera-se uma expressão de asserção um conjunto de instruções que retornam um valor a ser comparado na asserção (Figura 6).

Figure 6: Exemplo de uma expressão de asserção

```
ArrayList list = new ArrayList();
assertEquals(0, list.size());
```

No exemplo da Figura 6, a expressão da asserção *assertEquals* é a expressão *list.size()*.

2.2. ADEQUAÇÃO DE TESTES DE SOFTWARE

Desde as primeiras publicações sobre testes de software, a grande problemática da pesquisa nesta área foi responder a questão do que é um teste completo, ou seja, qual é o conjunto de casos de teste que garante que um programa está livre de erros (GOODENOUGH, 1975).

As primeiras tentativas de resolver esse problema recorreram à definição de testes exaustivos, ou seja, testes que exercitassem a combinação de todas as variações de entrada possíveis em um programa. No entanto, tais tentativas logo esbarraram nos limites dos recursos computacionais (ZHU et al, 1997; MYERS, 2004). A partir daí, os pesquisadores passaram a buscar o subconjunto de casos de testes, entre todos os possíveis, que garantisse um nível de confidência no programa sendo testado (MYERS, 2004). Diferentes técnicas de testes foram então propostas para tentar encontrar o melhor subconjunto de casos de teste, baseados em características de efetividade (capacidade de capturar erros) e custo computacional.

É nesse contexto que surgem os critérios de adequação de testes (*test adequacy criteria*). Estes critérios definem mecanismos de validação das próprias suítes de teste. Têm como objetivo medir a capacidade do teste em encontrar um erro no programa, ou seja, validar o quanto o teste está adequado ao sistema sendo testado (GOODENOUGH, 1975). Em uma analogia grosseira poderiam ser chamados de "teste do teste".

2.2.1 Classificação de critérios de adequação

Segundo Zhu et al (1997), a adequação de testes pode ser classificada de duas maneiras: de acordo com a fonte de informação pela qual são guiadas e de acordo com a abordagem de teste utilizada.

2.2.1.1 Critérios de adequação de acordo com a fonte de informação

Uma das formas de classificar critérios de adequação é de acordo com a fonte de informação usada para especificar os requisitos do teste. Nesta forma é possível classificar critérios de adequação em dois grupos: baseados na especificação ou baseados no programa.

No primeiro grupo, a fonte de informação para o teste é a especificação do sistema. Nessa abordagem um teste é considerado adequado se assegura todas as funcionalidades identificadas na especificação.

No segundo grupo, os testes são guiados de acordo com as construções do programa em si. Um programa é considerado adequado se todas as suas construções são exercitadas a partir dos testes. Essas construções variam de acordo com o critério proposto e podem ser desde linhas de código, instruções, condições, caminhos ou mesmo a combinação entre elas.

2.2.1.2 Critérios de adequação de acordo com a abordagem de teste

Ainda de acordo com Zhu et all (1997), há uma outra classificação possível para critérios de adequação: a abordagem de teste utilizada. Nessa classificação existem três grupos possíveis de critérios de adequação: baseados em testes estruturais (*structural testing*), baseados em falhas ou defeitos (*fault-based testing*) e baseados em erros (*error-based testing*).

2.2.1.2.1 Adequação de testes baseada em abordagem estrutural de testes

Em testes estruturais, os requisitos do teste são especificados de acordo com os elementos estruturais do programa ou da especificação. A adequação consiste em atingir a cobertura desses elementos. Os diferentes critérios de cobertura de código, como cobertura de linhas de código e cobertura de caminhos, bem como a cobertura de estados entram nesse grupo.

2.2.1.2.2 Adequação de testes baseada em falhas

A adequação de testes baseados em falhas consiste em medir a capacidade do teste em detectar falhas. O critério de adequação mais comum nesse grupo é a análise de mutação. Essa técnica consiste na injeção sistemática de falhas no código fonte ou objeto do programa com o objetivo de mensurar o quanto de falhas uma suíte de testes consegue capturar. A inserção dessas falhas se dá através da modificação de estruturas do programa, como condições, atribuições e laços, os denominados operadores de mutação (*mutation operators*). Um programa modificado é considerado um programa mutante. Ao executar a suíte de testes, é feita uma contagem dos mutantes que foram "capturados". Um mutante é "capturado" se o teste falhar por influência dele. O total de mutantes não equivalentes capturados é chamado *mutation score* ou *mutation adequacy*. Um mutante é considerado equivalente se sua semântica é considerada igual ao programa original, ou seja, computa o mesmo resultado. Saber de forma automatizada que dois programas são equivalentes é considerado um problema indecidível (KOSTER, 2007 e ANDREWS, 2005) e portanto se faz necessário a intervenção humana.

2.2.1.2.3 Adequação de testes baseada em erros

Já os testes baseados em erros consistem em identificar e testar pontos suscetíveis a erros dentro do programa. Considera-se como pontos suscetíveis os parâmetros de entrada do programa e as condições de ramificação que definem alterações no caminho de execução. Critérios de adequação baseados em erros consistem em identificar os subdomínios dos valores que alteram a execução do programa e gerar testes para as fronteiras entre esses subdomínios. A adequação se dá na razão entre os pontos críticos checados dentre todos os pontos críticos existentes.

2.3. COBERTURA DE CÓDIGO

Cobertura de código é um grupo de critérios de adequação de teste estruturais e baseados no programa. São critérios de adequação bastante populares por serem de fácil implementação e entendimento (MEYER et al, 2012; VANOVERBERGHE et al, 2012). A maioria dos critérios desse grupo são definidos por um fluxograma do programa (*flow-graph model*), onde as instruções são definidas por nodos no grafo e as mudanças no caminho de execução são definidas pelas arestas (ZHU et al, 1997).

2.3.1 Cobertura de linhas de código (Statement coverage)

Cobertura de linhas de código é um critério de cobertura de código onde a suíte de testes é considerada adequada se para todos os nodos do grafo existir pelo menos um teste que execute um caminho que contenha tal nodo. Apesar de bastante utilizado, por ser de fácil implementação e visualização, é considerado um critério bastante frágil em relação aos demais pois poucas variações nas entradas do programa podem atingir a adequação completa (ZHU et al, 1997).

2.3.2 Cobertura de ramos (Branch coverage)

Cobertura de ramos é um critério um pouco mais rigoroso do que cobertura de linhas de código. Para que seja considerado adequado de acordo com este critério, um teste deve conter caminhos de execução que contenham pelo menos uma passagem por cada aresta do fluxograma.

Um teste que esteja adequado de acordo com esse critério automaticamente estará adequado ao critério de linhas de código, pois se o teste passa por todas as arestas também estará executando todos as

linhas de código (ZHU et al, 1997). Essa relação, na qual um critério contém o outro, é denominado de subsumo entre critérios de adequação.

2.3.3 Cobertura de caminhos (Path coverage)

Ao tornar mais rigoroso o requisito de cobertura de ramos e exigir que todos os caminhos possíveis dentro do grafo sejam executados pelo menos uma vez (e não a simples execução das arestas) o teste estará adequado de acordo com o critério conhecido como cobertura de caminhos (ZHU et al, 1997).

Note que apesar de tornar os requisitos para que um teste esteja adequado mais rigoroso e portanto, pelo menos intuitivamente, aumentar a relação entre cobertura e qualidade do teste, esse estreitamento também torna mais difícil que as condições para uma cobertura completa sejam atingidas. Em alguns casos, inclusive, alguns critérios se tornam impraticáveis, devido a necessidade da computação de todos os caminhos possíveis dentro de um programa, geralmente crescendo exponencialmente em relação ao tamanho do código e possivelmente infinita se levar em consideração laços e funções recursivas (LU et al, 2006).

2.4. COBERTURA DE ESTADOS

Em 2007, Koster et al (2007) propuseram uma cobertura de código que levasse em conta não só a simples execução das estruturas do código mas também a execução dessas estruturas a partir das asserções do teste. Ao contrário das formas tradicionais de cobertura, para uma instrução ser considerada coberta nessa abordagem a mesma deve ser verificada a partir de uma asserção.

2.4.1 Definição de Koster para cobertura de estados

Koster et al (2007) definem como cobertura de estados um critério estrutural de adequação de testes (baseado no programa) que tem como objetivo medir a quantidade de verificações (asserções) que um teste faz na saída de um programa.

Um programa normalmente recebe parâmetros (*inputs*), faz o processamento desses parâmetros e devolve a resposta na forma de saídas (*outputs*). Um teste executa esse programa e faz verificações em suas

saídas através de asserções. A cobertura de estados irá medir a quantidade dessas saídas que são efetivamente verificadas por um teste.

Os autores entendem como *outputs* qualquer variável do programa que se mantém em memória após a execução do código do programa e antes da execução das asserções do teste. Isso inclui desde atributos das classes de produção (privados ou não) até retorno de métodos executados pelo teste (desde que permaneçam em memória). No entanto, variáveis locais de métodos, memórias desalocadas ou saídas do programa que não estejam em memória (escrita em arquivos, banco de dados ou rede) não são considerados nesta definição.

A última instrução do programa que modificou um *output* é considerada um *Output-Defining Statement* (ODS). Se um teste verificar um output então seu ODS estará coberto. Caso contrário, se um output não for verificado por nenhuma asserção o seu ODS não estará coberto. A cobertura de estados se dará pela quantidade de ODS verificados pelo teste dividido pela quantidade total de ODS do programa.

$$Cobertura\ de\ estados = \frac{Número\ de\ ODS\ cobertos}{Número\ de\ ODS\ do\ programa}$$

Como exemplo, considere o código listado na Figura 7.

Figura 7: Código destacando ODS's

```
public class Person {
   private int age = 0;
   public setAge(int age) {
      this.age = age; // ODS
   }
   public int getAge() {
      return this.age; // ODS
   }
}
@Test
public void testAge() {
  Person p = new Person();
  p.setAge(30);
  assertEquals(30, p.getAge());
}
```

No exemplo, a classe *Person* possui dois ODS's. Na primeira linha do método *setAge*, a atribuição no atributo *age* é a última instrução que o altera, portanto este é o primeiro ODS. O segundo é o retorno do método *getAge* que para Koster et al também é um *output* da classe. Como o teste *testAge* tem uma asserção no retorno do método *getAge* então está cobrindo este ODS. Por sua vez, o atributo age participa da expressão de retorno do método *getAge* e portanto seu ODS também estará coberto. Neste caso o teste estaria atingindo uma cobertura de estados completa, como mostra o cálculo abaixo.

$$Cobertura\ de\ estados = \frac{Número\ de\ ODS\ cobertos}{Número\ de\ ODS\ do\ programa} = \frac{2}{2} = 100\ \%$$

2.4.2 Definição de Vanoverberghe para cobertura de estados

Vanoverberghe et al (2012) propõem uma modificação à definição de cobertura de estados proposta por Koster et al. Em sua definição, Vanoverberghe et al propõem que o estado de um programa é o conjunto dos atributos de suas classes no momento das asserções (diferente de Koster que com sua noção de *outputs* incluía também o retorno de métodos). Além disso, Vanoverberghe et al baseia-se no conceito de *atualização de estados (state update)* como medida para o cálculo da taxa de cobertura de estados. Uma *atualização de estado* é equivalente ao ODS de Koster, ou seja, é a instrução na qual foi feita a última modificação em um atributo de um objeto. A diferença é que Vanoverberghe insere mais uma informação à atualização de estado: o identificador do objeto que teve seu atributo modificado.

Baseado nesta definição são propostos dois tipos de cobertura de estados: cobertura de estados insensitiva a objetos e cobertura de estados sensitiva a objetos.

2.4.2.1 Cobertura de estados insensitiva a objetos

Para a cobertura de estados insensitiva a objetos uma *atualização de estado* nada mais é do que a localização no código onde foi feita a última modificação de um atributo de uma classe. Corresponde ao ODS proposto por Koster.

2.4.2.2 Cobertura de estados sensitiva a objetos

Na definição de cobertura de estados sensitiva a objetos guarda-se o identificador do objeto que teve seu atributo modificado. A justificativa para isso, segundo os autores, é que um programa pode ter mais de um objeto de uma classe em execução durante um teste. Uma asserção pode verificar a modificação de um atributo de um dos objetos e esconder erros em outros objetos que não participaram de asserções.

Para este trabalho optou-se por usar a definição proposta por Vanoverberghe, porém utilizando uma cobertura de estados insensitiva a objetos. A justificativa para essa escolha está no fato de que as informações adicionais acabam acrescentando complexidade na leitura dos resultados do teste (por exigir a identificação de qual objeto o resultado está se referindo). Além disso acredita-se que há a possibilidade de que o testador acabe fazendo asserções redundantes apenas para melhorar o nível de cobertura do teste.

2.4.3 Cálculo de cobertura de estados

O cálculo da cobertura de estados deve considerar quais atributos foram modificados pelo teste e quais desses atributos estão cobertos por asserções.

2.4.3.1 Estado modificado

O estado de uma classe é modificado quando alguma instrução do programa altera o valor de algum de seus atributos. Em linguagens de programação essa instrução normalmente corresponde a uma atribuição. No exemplo da Figura 8 o atributo *age* está sendo modificado, devido à atribuição no método *setAge*.

Figura 8: Exemplo de modificação de um atributo

```
public setAge(int age) {
   this.age = age;
}
```

Este tipo de modificação talvez seja o mais simples de se identificar. No entanto, é comum em programas orientados a objetos o

uso de atributos de tipos compostos (estruturas de dados) como listas, conjuntos e dicionários. A modificação destes atributos pode não existir apenas na forma de atribuições mas também através de alterações em sua estrutura interna, como inserções, remoções e alterações em seus elementos. Considere o exemplo da Figura 9.

Figura 9: Exemplo de modificação em atributos de tipo composto

```
public class Polygon {
   private List<Point> points =
                    new ArrayList<Point>();

   public void addPoint(Point point) {
      this.points.add (point);
   }
   public int pointCount() {
      return points.size();
   }
}
@Test
public void testAddPoint() {
   Polygon polygon = new Polygon();
   polygon.addPoint(new Point(0, 0));
   assertEquals(1, polygon.pointCount());
}
```

Se apenas atribuições forem consideradas modificações em atributos o método *addPoint* não estaria modificando o atributo *points* da classe *Polygon* e poderia não ser exposto à análise de cobertura de estados. Nenhum dos trabalhos existentes na literatura de cobertura de estados trata este caso. De fato, apresentar uma solução para este problema é uma das contribuições deste trabalho e será melhor explicada no Capítulo 4.

2.4.3.2 Estado coberto

O estado de uma classe está coberto quando há asserções no teste verificando os valores de cada um de seus atributos modificados. Não é necessário que essa verificação seja feita diretamente nos valores dos atributos, desde que estes influenciem o resultado da asserção.

2.4.3.2.1 Influência de um atributo em uma asserção

Um atributo influencia uma asserção quando o resultado da asserção depende do valor do atributo. Se ao modificar o valor do atributo o resultado da asserção também for alterado então pode-se dizer que o atributo está coberto.

Saber se um atributo influencia o resultado de uma asserção é um problema de análise de fluxo de dados de programas. O que se pretende saber é se os participantes da expressão de asserção (métodos, variáveis) são influenciados pelo valor do atributo em questão. Considere o exemplo da Figura 10.

Figura 10: Exemplo de influência de um atributo em uma asserção

```
public class Point {
   private double x, y;
   public Point(double x, double y) {
      this.x = x;
      this.y = y;
   }
   public double sumX(double xToSum) {
      double tmp = this.x + xToSum;
      return tmp;
   }
}

@Test
public void test() {
   Point point = new Point(10, 10);
   assertEquals(11, point.sumX(1));
}
```

Neste exemplo, o método *sumX* da classe *Point* é integrante da expressão de asserção. O método *sumX*, por sua vez, soma um valor recebido como parâmetro ao atributo *x*, guarda o resultado da soma em uma variável local *tmp* e retorna o seu valor. O valor do atributo *x* não está sendo verificado diretamente na asserção. No entanto o seu conteúdo tem participação no resultado do método *sumX* (este sim participante direto da asserção). Se o seu valor fosse modificado a asserção também teria que ser modificada. Quando isso acontece assume-se que o atributo

influenciou o resultado da asserção e portanto está coberto. Em contrapartida considere a situação exposta na Figura 11:

Figura 11: Exemplo de variável que não influencia o retorno do método

```
public double sumX(double xToSum) {
   double tmpY = this.y + xToSum;
   double tmp = this.x + xToSum;
   return tmp;
}
```

Neste exemplo, o código do método *sumX* é modificado, fazendo com que o atributo *y* também tenha seu valor somado ao parâmetro recebido no método. O resultado dessa soma também é guardado em uma variável local. No entanto, apesar do atributo *y* ser executado durante o teste o seu valor não influencia o resultado da asserção. Neste caso assume-se que o atributo *y* não está coberto.

Uma das contribuições deste trabalho é apresentar um novo algoritmo de verificação de influências entre métodos e atributos. Esse novo algoritmo será apresentado no Capítulo 4.

2.5 OBSERVAÇÃO SOBRE NOMENCLATURA

Apesar de cobertura de estados ser uma forma de cobertura de código, neste trabalho é referenciada através do termo “cobertura de estados”, enquanto as outras formas de cobertura tradicionais (linhas de código, ramos e caminho) são referenciadas através do termo “cobertura de código”. Essa distinção visa estabelecer uma melhor clareza de conceitos durante a leitura do texto.

3 REVISÃO BIBLIOGRÁFICA

3.1 METODOLOGIA DA PESQUISA

A revisão bibliográfica foi realizada através de uma revisão sistemática de acordo com a proposta apresentada por Kitchenham (2007). Inspirada em métodos de pesquisa originárias das áreas médicas, Kitchenham propõe diretrizes para uma revisão sistemática em engenharia de software, com adaptações para as especificidades da área.

Uma destas diretrizes é a realização de um planejamento da pesquisa através de um protocolo de pesquisa sistemática. Tal protocolo tem como objetivo encontrar termos de busca não-ambíguos que sirvam como parâmetro para a reprodução da pesquisa por qualquer pessoa. Esses termos são formados a partir de perguntas definidas no protocolo.

A seguir é exposto o protocolo de pesquisa utilizado neste trabalho.

3.1.1 Protocolo de pesquisa sistemática

3.1.1.1 Objetivos da pesquisa

Esta pesquisa tem dois objetivos específicos. O primeiro é investigar se existem na literatura de testes de software outros trabalhos citando problemas com cobertura de código e caso existam, quais são as soluções propostas por eles. O segundo objetivo é levantar o estado da arte em cobertura de estados.

3.1.1.2 Questões da Pesquisa

Baseado nos objetivos da pesquisa expostos acima foram levantadas duas questões de pesquisa:

1. Existem trabalhos na literatura citando problemas com cobertura de código? Para estes trabalhos, quais são as soluções propostas por eles?
2. Quais são os trabalhos existentes sobre cobertura de estados?

3.1.1.3 Termo de busca

Após a análise das questões de pesquisa extraiu-se os seguintes termos de busca:

1. “software testing” and ("code coverage" or "test coverage") and (problem or misuse or effectiveness)
2. “software testing” and “state coverage”

3.1.1.4 Critério de seleção dos artigos:

Serão excluídos artigos que tratem dos seguintes temas:

- Teste baseado em estados (*state-based testing*) e testes baseados em modelo (*model-based testing*).
- Estudos de caso de uso de cobertura de código
- Priorização de casos de testes a partir de cobertura de código
- Cobertura em projetos de hardware
- Cobertura de código a partir de testes de interface
- Estudos de minimização de casos de testes através de cobertura de código.
- Cobertura de testes baseados nos requisitos de software (testes funcionais)

Serão incluídos apenas artigos em inglês.

3.1.1.5 Fontes de pesquisa

As principais fontes de pesquisa utilizadas neste trabalho foram bibliotecas digitais de artigos científicos. As bibliotecas digitais utilizadas estão listadas no quadro 1.

Quadro 1: Listagem das bibliotecas digitais utilizadas

Nome da biblioteca	Endereço da biblioteca
IEEExplore	http://ieeexplore.ieee.org
ACM Digital Library	http://portal.acm.org

3.1.1.6 Estratégias para a busca de estudos primários

A sequência de procedimentos utilizada para a realização da busca está listada a seguir.

1. Para cada uma das bibliotecas digitais escolhidas executa-se a pesquisa utilizando o termo de busca. Nessa primeira etapa serão excluídos apenas os artigos que não fazem parte do tema e foram trazidos pela busca por algum engano semântico. Essa análise será feita lendo o seu resumo. Serão colhidas as primeiras 5 páginas por termo.
2. Uma segunda seleção é feita baseada na leitura parcial do artigo (resumo e conclusão), julgando-o relevante ou não para a pesquisa (baseando-se no critério de seleção apresentado no item 3.1.1.4).
3. Para cada artigo encontrado no passo 2 são analisadas suas referências em busca de novos artigos.
4. Repete-se o passo 2 para cada novo artigo encontrado.

3.1.1.7 Estratégias de extração de dados

A extração das referências será feita manualmente através da cópia do arquivo no formato *bibtex (2015)* disponibilizado pelas bibliotecas digitais. Serão catalogados no software gerenciador de referências JabRef (2013).

3.2. PESQUISA

Após a realização da pesquisa foram selecionados os artigos listados no Quadro 2.

Quadro 2: Listagem dos trabalhos encontrados por pergunta

Questões de pesquisa	**Publicações**
Pergunta 1: Existem trabalhos na literatura citando problemas com cobertura de código? Para estes trabalhos, quais são as soluções propostas por eles?	CHILENSKI et al (2002), GOODENOUGH et al (1975), DEL FRATE et al (1995), CAI et al (2005), STOBIE et al (2005), INOZEMTSEVA et al (2014), MIRANDA (2014), GAY et al (2014), NAMIN et al (2009), MEYER et al (2012), KANER (2003), MARICK (1999), BHANSALI (2007)
Pergunta 2: Quais são os trabalhos existentes sobre cobertura de estados?	KOSTER et al (2007), VANOVERBERGUE et al (2012), KOSTER (2008), DUESTERWALD et al (1992)

3.2.1. Problemas com cobertura de código

Pergunta 1: Existem trabalhos na literatura citando problemas com cobertura de código? Para estes trabalhos, quais são as soluções propostas por eles?

Apesar de ser um critério de adequação bastante utilizado, principalmente pela sua facilidade de entendimento e visualização, a cobertura de código é também motivo de bastante controvérsia entre os pesquisadores da área. Muitas pesquisas colocam em dúvida se existe relação entre qualidade de testes e cobertura de código (INOZEMTSEVA e HOLMES, 2014).

Estes questionamentos se tornaram mais fortes na medida em que diretrizes para o desenvolvimento de sistemas críticos, como sistemas de aviação e satélite, passaram a adotar estas métricas como requisitos para a aceitação de seus softwares (CHILENSKI et al, 2002).

Desde o início das pesquisas nessa área já se sabia que o fato do teste estar executando todos as linhas, ramos ou caminhos tinha pouca relação com a qualidade de um programa. Goodenough e Gerhart (1975), em uma das primeiras publicações sobre adequação de testes, já falavam dessa baixa relação. Os autores apresentaram um programa incorreto (com erro na implementação) e construíram uma tabela de decisão, selecionando dados de teste que satisfaziam diferentes critérios de

cobertura. Como nenhum dos critérios foi capaz de evidenciar os erros no programa, concluíram que testes baseados apenas nas estruturas do programa não foram suficientes para provar que este não continha erros.

Para alguns autores é possível notar uma correlação positiva entre cobertura de código e capacidade de detectar erros (DEL FRATE et al, 1995). Um programa que tenha todas as suas linhas executadas durante o teste possui mais chances de ter erros encontrados do que um que não tenha nenhuma linha executada (CAI et al, 2005). No mínimo assegura-se que o programa não termina de maneira inesperada, causado por algum acesso indevido de memória, por exemplo.

No entanto, a simples execução das linhas não permite afirmar que o programa está se comportando como o especificado (STOBIE, 2005). Inozemtseva e Holmes (2014) buscaram essa correlação através de uma experiência com 31000 suítes de teste em cinco sistemas diferentes, consistindo em mais de 724000 linhas de código (a maior base de testes para este tipo de experiência já encontrada na literatura). A conclusão que chegaram é que há pouca correlação (entre baixa e moderada) entre cobertura de código e eficácia do teste.

Miranda (2014) atenta para o problema de que as métricas de cobertura "tradicionais" não são mais capazes de atender às demandas impostas no escopo do surgimento de novos paradigmas de programação, como programação orientada a componentes e serviços. Para ele, uma revisão destas métricas com base neste novo contexto é necessária. Para isso propõe uma nova definição de cobertura de código denominada cobertura relativa. Nessa definição uma suíte de testes não precisa necessariamente executar todas as construções existentes no programa mas apenas aquelas que são de interesse para um determinado pedaço do programa (um componente, por exemplo).

Gay et al (2014) sugere recomendações para melhorar a eficácia de uma suíte de testes no contexto de geração automática de testes. Os autores concordam que não é suficiente que uma suíte de testes tenha 100% de cobertura de código para que uma ferramenta de geração automática de testes considere o programa testado. É necessário, segundo os autores, um maior rigor das métricas de cobertura usadas para este fim.

Namin et al (2009) investiga a correlação entre tamanho da suíte de testes e cobertura de código na eficácia de uma suíte de testes. O objetivo é saber se a correlação encontrada por alguns pesquisadores entre aumento da cobertura de código e capacidade de detectar bugs em um programa está relacionado com o aumento da cobertura em si ou ao aumento do tamanho da própria suíte de teste. Para isso foram feitos experimentos nos quais o tamanho da suíte de testes se mantinha

constante variando apenas a sua cobertura. Na sequência foi feito o inverso: o tamanho da suíte variava enquanto a cobertura se mantinha constante. Para os dois experimentos notou-se uma correlação (entre baixa e moderada) entre as variáveis em questão e a eficácia dos testes, mostrando que tanto o tamanho quanto a cobertura das suítes de teste exercem influência moderada na capacidade do teste de capturar *bugs*, uma independente da outra.

Meyer et al (2012) faz um estudo empírico executando testes randômicos em programas por 2520 horas. Durante esse período foram coletadas informações sobre a cobertura de ramos atingida pelos testes e a quantidade de bugs não detectados por estes. Os resultados encontrados mostraram que a cobertura de ramos não é uma boa métrica para medir a eficácia de um teste.

Kaner (2003) nega a ideia de um teste completo como proposto por Goodenough e Gerhart (1975), sugerindo que um teste completo é impossível de ser definido. Segundo ele, para que um teste seja completo deve testar todas as combinações de entrada do programa, todos os caminhos de execução, todas as formas do usuário interagir com o sistema e todas as combinações de hardware nos quais o sistema poderá ser executado. Kaner faz duras críticas ao modelo de cobertura de código, pois ao tentar simplificar um teste completo, a cobertura de código abre espaço para implementações de testes fracos, objetivando apenas o alcance da métrica de cobertura.

Marick (1999) cita alguns casos em que a cobertura de código pode fornecer falsos positivos para o testador, como por exemplo erros de falta de código (*fault of omission*), uma classe de erros onde o problema não está no código existente mas sim na falta de código para tratamentos de casos especiais como condições lógicas, retornos de funções e exceções não tratadas.

Bhansali (2007) apresenta o problema do uso de métricas de cobertura de código em diretrizes para o controle de qualidade em sistemas críticos de aviação. Segundo os autores, para atingir a norma de qualidade do setor de aviação conhecida como DO-178B - Software Considerations in Airborne Systems and Equipment Certification (CHILENSKI et al, 2002) um programa deve estar adequado segundo o critério de cobertura MC/DC (*Modified Condition/Decision Coverage*). No entanto, como visto até agora e como exposto pelos autores, não há uma correlação entre tais métricas e a eficácia da suíte de testes, o que faz gerar uma falsa sensação de segurança no desenvolvimento de tais sistemas.

3.2.1.1 Observação sobre os problemas com cobertura de código

A pesquisa partiu da hipótese de que haviam problemas com cobertura de código tentando encontrar trabalhos na literatura que embasassem esta afirmação. Cobertura de estados surge como solução para alguns destes problemas (testes sem asserções ou asserções fracas). No entanto, é importante ressaltar que muitos dos problemas encontrados para cobertura de código também se aplicam para cobertura de estados e outros critérios estruturais de adequação (problemas com omissão de código, por exemplo).

3.2.2. Cobertura de estados

Pergunta 2: Quais são os trabalhos existentes sobre cobertura de estados?

Koster et al (2007) foram os primeiros a definir cobertura de estados. Segundo os autores, cobertura de estados é a razão entre as saídas de um programa verificadas por asserções e todas as saídas do programa. Como saída do programa entende-se variáveis que se mantém em memória durante a execução das asserções. Para saber se uma variável influencia uma asserção foi utilizado a técnica de *program slice* (WEISER, 1981).

Koster (2008) propõe uma ferramenta para o cálculo de cobertura de estados através da implementação de uma extensão ao JUnit (2015). Esta ferramenta faz a verificação de quais variáveis influenciaram uma asserção através do uso de *dynamic taint analysis* (HALDAR et al, 2005). No entanto, esta ferramenta não está disponível para uso.

Esse conceito de variáveis e instruções que influenciam a saída de um programa já tinha sido proposto por Duesterwald et al (1992) mas no contexto de cobertura de código tradicional.

Vanoverberghe et al (2012) apresentam uma nova definição de cobertura de estados baseada no conceito de atributos de objetos. Para eles o estado de um programa é o conjunto de atributos dos objetos vivos durante a execução das asserções. Essa definição é mais restritiva do que a de Koster (2008) por não considerar retorno de métodos públicos e parâmetros passados por referência como participantes do estado de um programa. Além disso Vanoverberghe et al propõem duas formas de cobertura de estados: sensitiva a objetos e insensitiva a objetos. A

primeira assume que diferentes objetos de uma mesma classe devem ter seus estados cobertos separadamente, enquanto que a segunda não exige essa diferenciação, bastando apenas que um objeto da classe tenha seus atributos verificados por asserções para que seu estado seja considerado coberto.

3.2.2.1 Observação sobre a exclusão de trabalhos sobre estratégia de teste baseado em estados

Teste baseado em estados (state-based testing) é uma estratégia de geração e seleção de testes de programas orientados a objetos que visa gerar testes que reflitam as diferentes transições de estados possíveis em um programa (TURNER et al, 1993). Nesta abordagem se faz necessário um mapeamento de todos os estados considerados significativos para o programa e as diferentes entradas do programa que geram estes estados. Um programa é considerado coberto se existirem casos de testes para cada um destes estados. A abordagem de cobertura de estados proposta por Koster et al (2007) se difere desta por ser um critério de adequação de testes estruturais, ou seja, possui como medida de adequação as estruturas do código do programa. No caso, da cobertura de estados proposta por Koster et al (2007) a medida de adequação são as modificações de estados executadas pelos testes que foram verificadas por asserções. Enquanto a definição de Koster et al (2007) tem como objetivo verificar se uma suíte de testes está fazendo asserções nas mudanças de estados de um programa, a cobertura para a estratégia de teste baseado em estados tenta saber se há casos de testes que executem todos os estados do programa.

4 UM ALGORITMO PARA O CÁLCULO DE COBERTURA DE ESTADOS

Este capítulo apresenta o algoritmo para o cálculo de cobertura de estados proposto nesta dissertação. Inicialmente são feitas definições sobre os conceitos propostos, necessários para o entendimento do algoritmo. Explica-se o que é a modificação e verificação de estados para este trabalho, demonstrando um novo algoritmo de verificação de influências entre atributos e métodos. Na sequência, o algoritmo de cálculo de cobertura de estados é apresentado, bem como o seu uso em um exemplo fictício.

4.1 DEFINIÇÕES

4.1.1 Estado de um programa

Cobertura de estados consiste na análise das modificações de estado de um programa feitas por um teste e de quais destas modificações estão sendo verificados por asserções.

Como exposto no Capítulo 2 não há um consenso na literatura de cobertura de estados sobre o que é o estado de um programa. Koster et al (2007) define como estado todas as variáveis vivas no momento da asserção. Isso inclui não só os atributos das classes mas também retorno de métodos.

Já Vanoverberghe et al (2012) simplifica dizendo que o estado de um programa são os valores dos atributos de suas classes em determinado ponto de execução. De acordo com esta definição, ao exercitar uma classe, um teste modifica os seus atributos. Tais atributos serão considerados o estado modificado da classe e, portanto, deverão ser verificados pelo teste através de asserções.

Neste trabalho optou-se pela definição de Vanoverberghe et al por ser mais simples e ainda assim abranger os casos mais importantes.

4.1.2 Modificação de estados

Para calcular a cobertura de estados é necessário saber quando um atributo de um objeto é modificado por um teste. Para este trabalho considera-se que os atributos podem ser alterados de duas maneiras:

1. Através da atribuição de valores

2. Através da manipulação de elementos em tipos compostos (estruturas de dados)

4.1.2.1 Atribuição de valores

A forma mais simples de visualizar a modificação de um atributo é quando este é o alvo (lado esquerdo) de uma expressão de atribuição. Se esta expressão é executada por um teste então considera-se que o atributo foi modificado por ele. Considere o exemplo da Figura 12.

Figure 12: Exemplo de modificação de atributos por atribuição de valores

```
class Point {
   private double x, y;

   public Point(double x, double y) {
      this.x = x;
      this.y = y;
   }
}

@Test
public void testPointConstruct() {
   Point point = new Point(10, 10);
   (...)
}
```

Neste exemplo a primeira e a segunda linha do construtor da classe *Point* são atribuições em *x* e *y*, respectivamente. Quando este construtor for executado a partir do teste *testPointConstruct* os atributos *x* e *y* serão modificados por ele.

4.1.2.2 Manipulação de tipos compostos (estruturas de dados)

Em linguagens modernas de programação orientadas a objetos é raro um programa que não se utilize de estruturas de dados como listas, vetores e dicionários disponibilizadas por bibliotecas padrão da linguagem. Na linguagem Java, por exemplo, são as estruturas de dados contidas do pacote *java.util* (*java.util.ArrayList*, *java.util.Map*, *java.util.Set*, etc).

Para o cálculo da cobertura de estados isso se torna um problema pois muitas vezes estas estruturas de dados são utilizadas como atributos das classes em teste. É de interesse do testador visualizar estes atributos no relatório de cobertura de estados. Mas como saber quando uma estrutura de dados é modificada?

Para resolver este problema é proposto uma segunda forma de identificar modificações de atributos quando estes são tipos compostos. Esta forma baseia-se na ideia de que uma estrutura de dados é modificada quando o seu conteúdo é modificado, ou seja, quando há chamadas para métodos de inserção, remoção e modificação de seus elementos (Figura 13).

Figura 13: Exemplo de modificações em um ArrayList

```
ArrayList<String> list = new ArrayList<String>();
list.add("1");
list.add("2");
list.remove(0);
```

A estes métodos que modificam estruturas de dados deu-se o nome de *métodos modificadores*. Como não é possível saber todos os *métodos modificadores* em uma biblioteca optou-se por uma solução de *whitelist* (ZHANG, 2015).

Nesta solução, é criada uma listagem de *métodos modificadores* para determinada biblioteca. Se durante a execução dos testes uma estrutura de dados executa um desses métodos listados então ela será considerada modificada pelo teste. Considere o exemplo da Figura 14.

Figura 14: Exemplo de modificação de estruturas de dados

```
class Polygon {
   public List<Point> points =
                     new ArrayList<Point>();

   public void addPoint(Point point) {
      this.points.add(point);
   }
}

@Test
public void test() {
   Polygon polygon = new Polygon();
   polygon.addPoint(new Point(10, 10));
   assertEquals(1, polygon.points.size());
}
```

Neste exemplo, como o método *java.util.ArrayList.add* está contido na *whitelist* de métodos modificadores de *java.util.ArrayList* (Figura 15) então o atributo *points* da classe *Polygon* é considerado um atributo modificado pelo teste.

Figura 15: Whitelist de métodos modificadores para a classe java.util.ArrayList

```
java.util.ArrayList.add
java.util.ArrayList.addAll
java.util.ArrayList.clear
java.util.ArrayList.remove
java.util.ArrayList.removeAll
java.util.ArrayList.removeRange
java.util.ArrayList.retainAll
java.util.ArrayList.set
```

No Capítulo 5 é apresentada uma implementação do algoritmo proposto no qual a listagem de métodos modificadores para a biblioteca padrão Java é disponibilizada.

4.1.2.3 Expressão modificadora

Para facilitar o entendimento do algoritmo ambas as formas de modificação de atributos expostas acima (modificação por atribuição e

modificação por *whitelist*) serão referenciadas no texto pelo termo *"expressão modificadora"*.

4.1.3 Verificação de estados

Após saber os atributos modificados pelo teste é preciso saber quais deles foram verificados por asserções. A maneira mais simples de um atributo ser verificado é quando o seu valor participa diretamente no valor da expressão de asserção. Considere o exemplo da Figura 16.

Figura 16: Exemplo de atributo público sendo verificado por uma asserção

```
class Point {
   public double x, y;

   public Point(double x, double y) {
      this.x = x;
      this.y = y;
   }
}

@Test
public void testPointConstruct() {
   Point point = new Point(10, 10);
   assertEquals(10, point.x);
}
```

No exemplo da Figura 16, pelo fato do atributo *x* ser público é possível fazer uma asserção diretamente no seu valor. Nesse caso, este será considerado verificado pelo teste. Nota-se, porém, que isso é uma má prática em programas orientados a objetos, onde o encapsulamento de atributos é um princípio importante. Normalmente, para que um cliente da classe tenha acesso aos seus atributos utiliza-se métodos de acesso para ler os seus valores. O atributo em si seria inacessível externamente através do uso de modificadores de acesso privado. O exemplo da Figura 17 demonstra esta situação:

Figura 17: Exemplo de atributo privado sendo verificado por uma asserção

```
class Point {
   private double x, y;
   (...)
   public double getX() {
       return this.x;
   }
}

@Test
public void testPointConstruct() {
   Point point = new Point(10, 10);
   assertEquals(10, point.getX());
}
```

Neste caso, não é mais possível fazer uma análise simples de quais atributos foram verificados pela asserção. É preciso saber antes quais atributos influenciaram no retorno do método *getX*, para considerá-los cobertos pelo teste.

Outra situação não trivial é quando um atributo causa influência na asserção não pelo valor de seu conteúdo mas sim por participar em alguma estrutura de decisão (*if* e *else*, por exemplo) que altere o resultado da asserção. É o que acontece no exemplo da Figura 18 com os atributos *x* e *y*. Ambos estão cobertos pela asserção pois influenciam no seu resultado.

Figura 18: Exemplo de atributos influenciando logicamente

```
class Point {
   private double x, y;
   (...)
   public boolean isZero() {
       if (this.x == 0 && this.y == 0)
          return true;
       else return false;
   }
}
@Test
public void testPointConstruct() {
   assertTrue(point.isZero());
}
```

Para resolver estes problemas, é proposto um novo algoritmo de verificação de influências entre atributos e métodos.

4.1.3.1 Algoritmo de verificação de influência entre atributos e métodos

O algoritmo proposto consiste em descobrir quais atributos de uma classe tiveram influência no resultado de um de seus métodos. Para isso utiliza-se do conceito de dependências entre identificadores.

4.1.3.1.1 Dependência entre identificadores

Um identificador é qualquer variável local, parâmetro, atributo ou método que faça parte do programa.

Um identificador é considerado dependente de outro em uma determinada expressão do código se o valor do primeiro for modificado em função do segundo.

No código da Figura 19, *a* é considerado dependente de *b* no momento da execução da instrução *a = b*.

Figura 19: Exemplo de dependência entre identificadores

```
int func() {
   int a = 0;
   int b = 10;
   a = b;
   return a;
}
```

Outra forma de dependência é quando um identificador participa de uma decisão lógica do programa que influencia outro identificador. No código da Figura 20, a condição da qual *b* participa influencia o valor de *a*. Portanto, diz-se também neste caso que *a* depende de *b*.

Figura 20: Exemplo de dependência lógica entre identificadores

```
int func() {
   int a = 0;
   int b = 10;
   if (b > 0);
      a = 5;
   return a;
}
```

4.1.3.1.2 Conjunto de dependências

O conjunto de dependências para um identificador *x* é formado por todos os identificadores aos quais *x* depende em determinado momento da execução do programa. No exemplo da figura 19, após a execução da terceira instrução o conjunto de dependências do identificador *a* será composto pelo identificador *b* (Figura 21)

Figure 21: Conjunto de dependências de *a*

```
Dep(a) = { b }
```

Esta análise de dependências é feita para cada instrução do método, inclusive para a instrução de retorno. A diferença é que para esta última as dependências encontradas na expressão de retorno serão adicionadas ao conjunto de dependências do próprio método (Figura 22).

Figure 22: Conjunto de dependências de um método

```
Dep(func) = { a }
```

Para cada identificador encontrado na expressão de retorno do método adiciona-se recursivamente as dependências destes no conjunto de dependências do método.

4.1.3.1.3 Definição do algoritmo de verificação de influências

As influências de um método são o subconjunto de dependências do método que são atributos. A definição para o algoritmo de verificação de influências está descrita abaixo.

Seja *m* um método. Seja *p* uma pilha de conjuntos de dependências.

1. Para cada instrução *i* de *m* atualiza-se o conjunto de dependências para os identificadores envolvidos de acordo com as seguintes regras:

    a. Seja *i* uma condição. Seja *C* o conjunto de identificadores contidos na expressão da condição *i*. Adiciona-se *C* na pilha de conjunto de dependências *p*.

    b. Seja *i* o final do escopo de uma condição. Remove-se o conjunto de identificadores da expressão da condição *i* da pilha *p*.

    c. Seja *i* uma expressão modificadora no formato $x = y$, onde *x* é um identificador e *y* uma expressão. Seja *C* o conjunto de identificadores contidos na expressão *y*.

        i. Para cada elemento c_i de *C* adiciona-se c_i no conjunto de dependências de *x*.
        ii. Para cada conjunto contido na pilha de conjuntos de dependências p adiciona-se seus elementos no conjunto de dependências de x.

    d. Seja *i* a expressão de retorno do método *m*. Seja *C* o conjunto de identificadores contidos em *i*. Para cada elemento c_i de *C* adiciona-se c_i no conjunto de dependências de *m*.

2. Seja *Dep(m)* o conjunto de dependências de *m*.

    a. Para cada elemento *x* de *Dep(m)* adiciona-se as dependências de *x* em *Dep(m)*.
    b. Procede-se recursivamente para cada novo elemento adicionado até que não haja mais elementos novos.

3. Seja *Attr(Dep(m))* o subconjunto de elementos de *Dep(m)* que são atributos. Este será o conjunto de atributos influentes de um método.

Nas Figuras 23 e 24 o algoritmo é apresentado através de pseudocódigo.

Figura 23: Pseudocódigo para o algoritmo de verificação de influências

```
function influencesOf is:
input: method M
output: set of influences of M

    let dep(M) be set of dependencies of M
    let instructions be instructions of M
    let scopeDeps be a stack of sets of dependencies

    for each instruction in instructions:

        if instruction is a decision expression
            let identifiers be list of identifiers of instruction
            append identifiers to scopeDeps

        if instruction is the end of a decision expression
            let identifiers be list of identifiers of instruction
            remove identifiers of scopeDeps

        if instruction is a modifier expression:
            let x be left side of instruction
            let y be right side of instruction
            let dep(x) be list of dependencies of x

            for each identifier in y:
                append identifier to dep(x)
            for each set in scopeDeps:
                append elements of set to dep(x)

        if instruction is a return expression:
            let identifiers be list of identifiers of instruction
            for each identifier in identifiers:
                append identifier to dep(M)

    let resolved_deps be result of resolveDependencies(dep(M))
    let attributes be subset of resolved_deps that are attributes
    return attributes
```

O algoritmo recursivo de resolução de dependências está definido na função *resolveDependencies*, como mostra o pseudocódigo da Figura 24.

Figura 24: Pseudocódigo para o algoritmo de resolução de dependências

```
function resolveDependencies is
input: set D of dependencies to resolve
output: set of resolved dependencies

   let resolved_dependencies be a list of dependencies

   for each dep in D:
      append resolveDependencies(dep) to resolved_dependencies

  return resolved_dependencies
```

4.1.3.1.4 Exemplo de uso do algoritmo de influências

Para exemplificar o uso do algoritmo considere o código da Figura 25.

Figura 25: Exemplo de execução do algoritmo

```
public class Point
{
   private double x, y;

   public Point(double x, double y) {
      this.x = x;
      this.y = y;
   }

   public double getX() {
      return this.x;
   }
   public double getY() {
      return this.y;
   }

   public double dotProduct(Point other) {
      double dotX = other.getX() * this.getX();
      double dotY = other.getY() * this.getY();
      return dotX + dotY;
   }
}
```

Neste exemplo, deseja-se saber quais atributos da classe *Point* têm influência no resultado do método *dotProduct.* Para resolver isso é preciso descobrir o conjunto de dependências do método. Segundo o algoritmo proposto é preciso analisar cada uma das instruções individualmente.

1. Análise da primeira linha do método dotProduct

Na primeira linha do método, há uma atribuição na variável *dotX*, como mostra a figura 26.

Figura 26: Atribuição da variável *dotX*

```
double dotX = other.getX() * this.getX();
```

Como a expressão à direita contém o identificador *getX*, adiciona-se este último no conjunto de dependências da variável *dotX* (figura 27). Isso quer dizer que o conteúdo da variável *dotX*, neste momento da execução, depende do resultado do método *getX*.

Figura 27: Conjunto de dependências de dotX após execução da primeira linha do método dotProduct

$$dep(dotX) = \{ getX \}$$

2. Análise do método *getX*.

Como *getX* também é executado pelo teste o seu conjunto de dependências também precisa ser analisado. Como pode-se notar na figura 28 a única linha do método é o retorno do valor do atributo *x*.

Figura 28: Código do método getX

```
public double getX() {
   return this.x;
}
```

Como *x* participa da expressão de retorno então é adicionado ao conjunto de dependências de *getX* (Figura 29).

Figura 29: Conjunto de dependências do método getX

$$dep(\text{getX}) = \{ \text{x} \}$$

3. Análise da segunda linha de dotProduct

Voltando ao método *dotProduct*, a sua segunda linha apresenta a mesma estrutura de modificação da primeira, com a diferença de que o alvo da expressão é a variável *dotY* e o identificador dependente é o método *getY* (Figura 30).

Figura 30: Atribuição da variável *dotY*

```
double dotY = other.getY() * this.getY();
```

Após a execução desta linha o conjunto de dependências de *dotY* será acrescido do identificador *getY* (figura 31).

Figura 31: conjunto de dependências de dotY após execução da primeira linha do método dotProduct

$$dep(\text{dotY}) = \{ \text{getY} \}$$

4. Análise do método getY

O método *getY*, por sua vez, apresenta a mesma estrutura do método *getX*. O retorno do método é dependente apenas do atributo *y* (Figura 32).

Figura 32: Conjunto de dependências do método getY

$$dep(\text{getY}) = \{ \text{y} \}$$

5. Análise da terceira linha do método dotProduct

A terceira linha apresenta a expressão de retorno do método (Figura 33).

Figura 33: Terceira linha do método dotProduct

```
return dotX + dotY;
```

Como pode-se notar os participantes da expressão são os identificadores *dotX* e *dotY*. Por esse motivo os dois identificadores serão adicionados ao conjunto de dependências do método dotProduct (Figura 34).

Figura 34: Conjunto de dependências de dotProduct após execução de sua terceira linha

$$dep(\text{dotProduct}) = \{\, \text{dotX}, \text{dotY} \,\}$$

6. Resolução do cálculo de dependências

Como *dotX* e *dotY* também possuem suas próprias dependências estas serão adicionadas ao conjunto de dependências de *dotProduct*. Procede-se dessa maneira recursivamente para qualquer outro identificador que contenha suas próprias dependências, como mostra a Figura 35.

Figura 35: Resolução do cálculo de dependências para o método dotProduct

$$dep(\text{dotProduct}) = \{ \text{dotX}, \text{dotY} \}$$

Resolução de dependências para a variável dotX

$$dep(\text{dotX}) = \{ \text{getX} \}$$
$$dep(\text{getX}) = \{ \text{x} \}$$
$$dep(\text{dotX}) = \{ \text{x} \}$$

Resolução de dependências para o método dotY

$$dep(\text{dotY}) = \{ \text{getY} \}$$
$$\text{dep}(\text{getY}) = \{ \text{y} \}$$
$$dep(\text{dotY}) = \{ \text{y} \}$$

Resolução de dependências para o método dotProduct

$$dep(\text{dotProduct}) = \{ \text{dotX}, \text{dotY} \}$$
$$dep(\text{dotProduct}) = \{ \text{x}, \text{y} \}$$

A Figura 36 mostra o conjunto de dependências final para o método *dotProduct*.

Figura 36: Conjunto de dependências final para o método dotProduct

$$\text{dep}(\text{dotProduct}) = \{ \text{x}, \text{y} \}$$

Conclui-se com a análise que o método *dotProduct* é influenciado pelos atributos *x* e *y* da classe *Point*.

4.2 CÁLCULO DE COBERTURA DE ESTADOS

O algoritmo de verificação de influências é apenas uma parte do algoritmo geral para o cálculo de cobertura de estados. Esta seção apresenta a definição deste último.

O algoritmo proposto tem três passos distintos. O primeiro é identificar os atributos modificados pelos testes. O segundo é identificar os atributos cobertos pelas asserções destes testes. Para isso é preciso extrair os identificadores envolvidos nas expressões de asserções do teste. Para cada identificador envolvido resolve-se recursivamente suas dependências através do algoritmo de verificação de influências exposto na seção 4.1.3.1. Estes serão os atributos cobertos pelo teste. No terceiro passo, a taxa de cobertura para o teste é definida através da divisão entre o número de atributos cobertos pelo número de atributos modificados.

4.2.1 Definição do algoritmo

Passo 1: Identificação dos atributos modificados

Seja *E* uma expressão modificadora no formato $x = y$, seja x um atributo de uma classe e y uma expressão. Seja *T* um teste e $C^{modificados}$ o conjunto de atributos modificados em *T*. Se a expressão modificadora *E* é executada pelo teste *T* então $x \in C^{modificados}$.

Passo 2: Identificação dos atributos cobertos

Seja *T* um teste e $C^{cobertos}$ o seu conjunto de atributos cobertos. Seja *a* uma asserção em *T* e $C^{identificadores}$ o conjunto de identificadores de *a*. Para cada identificador *id* em $C^{identificadores}$ adiciona-se o conjunto de influências de *id* a $C^{cobertos}$.

Passo 3: Cálculo da taxa de cobertura de estados

Seja *T* um teste. Seja $C^{cobertos}$ o conjunto de atributos cobertos por *T*. Seja $C^{modificados}$ o conjunto de atributos modificados por *T*. Se c é o número de elementos de $C^{cobertos}$ e m o número de elementos de $C^{modificados}$. a taxa de cobertura para o teste *T* será c/m.

A figura 37 apresenta o algoritmo no formato de pseudocódigo.

Figura 37: Pseudocódigo para o algoritmo de cobertura de estados

```
function state_coverage  is:
input: program P, test T
output: double state coverage ratio

   let modified_attributes be set of modified attributes of T
   let covered_attributes be set of covered attributes of T

   for each instruction in P:

      if instruction is a modifier expression:
         let x be left side of instruction
         if x is an attribute
            append x to modified_attributes

   for each assertion in T:

      let identifiers be the identifiers of assertion expression
      for each id in identifiers
        let dep(id) be influences_of(id)
        append dep(id) to covered_attributes

  let total_covered_attributes be the size of covered_attributes
  let total_modified_attributes be the size of modified_attributes
  return total_covered_attributes / total_modified_attributes
```

4.3 EXEMPLO

Para facilitar o entendimento do algoritmo a sua aplicação será mostrada através de um exemplo a seguir. Considere para isso o código da Figura 38.

Figura 38: Exemplo de execução do algoritmo de cobertura de estados

```
public class Point
{
   private double x, y;

   public Point(double x, double y) {
      this.x = x;
      this.y = y;
   }
   public double getX() {
      return this.x;
   }
   public double getY() {
      return this.y;
   }

   public double dotProduct(Point other) {
      double dotX = other.getX() * this.getX();
      double dotY = other.getY() * this.getY();
      return dotX + dotY;
   }
}

@Test
public void testDotProduct() {
   Point point = new Point(10, 10);
   assertEquals(400,
      point.dotProduct(new Point(20, 20)));
}
```

No exemplo há um teste chamando *testDotProduct* testando a classe *Point*. Aplicando o algoritmo, é preciso inicialmente identificar os atributos modificados.

4.3.1 Passo 1: Identificação dos atributos modificados

A classe *Point* possui dois atributos (*x* e *y*), um construtor e três métodos (*getX*, *getY* e *dotProduct*). A primeira linha do teste *testDotProduct* instancia um objeto da classe *Point*, fazendo com que seu construtor seja executado. Durante a execução do construtor há alterações nos atributos x e y, como mostra a Figura 39.

Figura 39: Atribuição no construtor da classe

```
public Point(double x, double y) {
   this.x = x;
   this.y = y;
}
```

Como tais atribuições são executadas pelo teste então considera-se *x* e *y* pertencentes ao conjunto *modified_attributes* do teste (Figura 40).

Figure 40: Conjunto de atributos modificados pelo teste testDotProduct

$$modified\_attributes = \{ x, y \}$$

4.3.2 Passo 2: Identificação dos atributos cobertos

O segundo passo é identificar os atributos cobertos pelo teste. Para isso é preciso realizar dois procedimentos. Primeiro extrai-se os identificadores participantes de asserções. Na sequência resolve-se as influências destes identificadores recursivamente, inserindo-as no conjunto *covered_attributes*.

4.3.2.1 Extração dos identificadores das asserções

O teste apresenta apenas uma asserção. Trata-se do *assertEquals*, como mostra a Figura 41.

Figura 41: Asserção no teste *dotProduct*

```
assertEquals(400,
      point.dotProduct(new Point(20, 20)));
```

A asserção é feita em cima do resultado do método *dotProduct*, portanto este será o identificador ao qual se deseja verificar as influências.

4.3.2.2 Resolução de influências dos identificadores

Para cada identificador participante da expressão de asserção buscam-se os atributos que o influenciam adicionando estes ao conjunto *covered_attributes*.

No exemplo, o único identificador participante é o método *dotPoduct.* Os atributos que o influenciam foram calculados anteriormente na seção 4.1.3.1 e são apresentados novamente na Figura 42.

Figura 42: Atributos que influenciam o método dotProduct

$$influenceOf(\text{dotProduct}) = \{ \text{x}, \text{y} \}$$

Como o método *dotProduct* é participante da expressão de asserção do teste então todas as suas influências serão consideradas cobertas pela asserção. O conjunto de atributos cobertos pelo teste é mostrado na Figura 43.

Figura 43: Conjunto de atributos cobertos pelo método *dotProduct*

$$\text{covered\_attributes}(\text{testDotProduct}) = \{ \text{x}, \text{y} \}$$

4.3.3 Passo 3: Cálculo da taxa de cobertura de estados

Calculando a taxa de cobertura para o teste tem-se a seguinte expressão (Figura 44):

Figure 44: Cálculo da taxa de cobertura de estados

$$covered\_attributes = \{ \text{x}, \text{y} \}$$
$$modified\_attributes = \{ \text{x}, \text{y} \}$$

$$State\ Coverage = \frac{\text{size(covered\_attributes)}}{\text{size(modified\_attributes)}}$$

$$State\ Coverage = \frac{2}{2} = 1 = 100\%$$

Como todos os atributos modificados pelo teste também participaram de suas asserções então a taxa de cobertura de estados para ele é de 100%.

5 IMPLEMENTAÇÃO

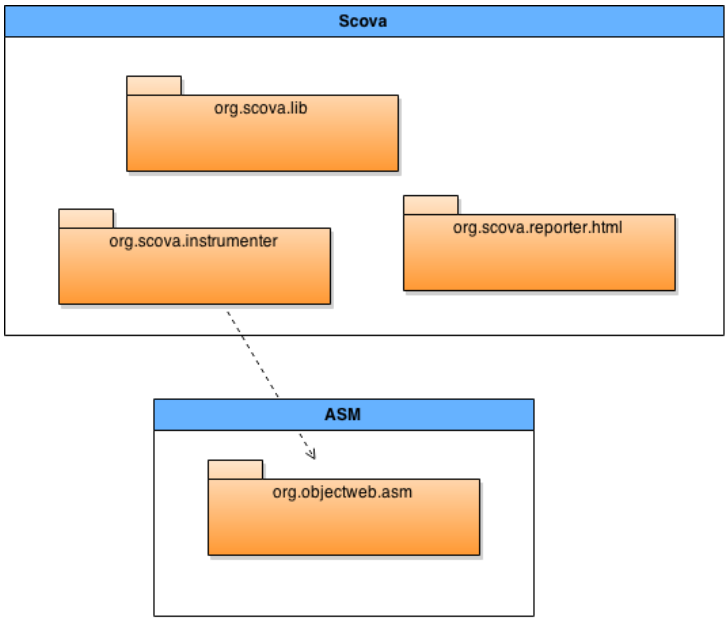
Foi realizada uma implementação do algoritmo na linguagem Java. Para esta implementação deu-se o nome de Scova e a mesma está disponível como código aberto (SCOVA, 2015).

Como o algoritmo proposto depende de informações do programa em tempo de execução optou-se por realizar uma instrumentação do *bytecode* Java. Instrumentação de código é uma técnica de análise de programas onde o código original é modificado com instruções artificiais com o objetivo de evidenciar informações contidas no código durante a sua execução. Para realizar a instrumentação foi utilizada a biblioteca ASM (BRUNETON, 2002).

5.1 ARQUITETURA DO SCOVA

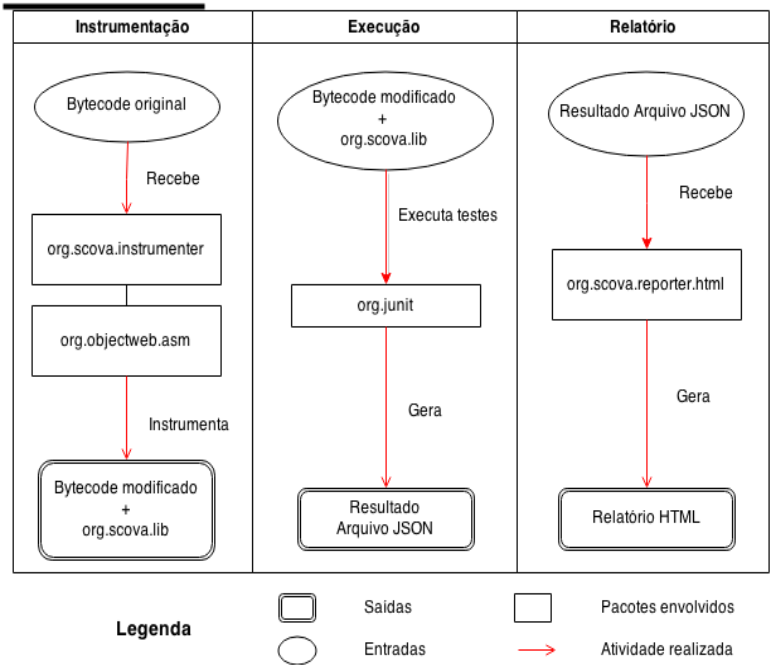
A arquitetura geral do SCOVA é mostrada na Figura 45.

Figura 45: Arquitetura geral do Scova



O diagrama da Figura 46 mostra a sequência de operações feitas nas diferentes fases do seu ciclo de vida (instrumentação, execução e relatório). Para cada fase o diagrama apresenta as entradas e saídas processadas pelos pacotes envolvidos.

Figura 46: Diagrama de ciclo de vida do Scova



Durante a fase de instrumentação o pacote *org.scova.instrumenter* recebe como parâmetro o *bytecode* compilado original do programa e com o auxílio da biblioteca ASM o modifica inserindo entre as instruções originais chamadas para métodos da biblioteca *org.scova.lib*. Essa biblioteca será *linkada* juntamente com o *bytecode* instrumentado. Durante a execução dos testes, as chamadas feitas para a biblioteca org.scova.lib coletam as informações necessárias para o cálculo de cobertura de estados. No final, um arquivo no formato JSON é criado para cada teste, contendo informações sobre os atributos modificados, atributos cobertos e a taxa de cobertura de estados. Na última fase o pacote *org.scova.reporter.html* interpreta esses arquivos e gera um relatório no formato HTML.

5.2.1 Relatórios

A partir dos arquivos JSON criados é possível criar relatórios para diferentes plataformas. Para o presente trabalho foram criados dois tipos de relatórios. Um que mostra os resultados em formato HTML (Figura 47) e um *plugin* pro Eclipse (Figura 48). Ambos estão disponíveis no mesmo repositório do Scova.

Figura 47: Relatório Html apresentado pelo Scova

State Coverage Report

Total State Coverage

Total Modified	Total Covered	Total State Coverage
6	4	0.6666666666666666

Global report

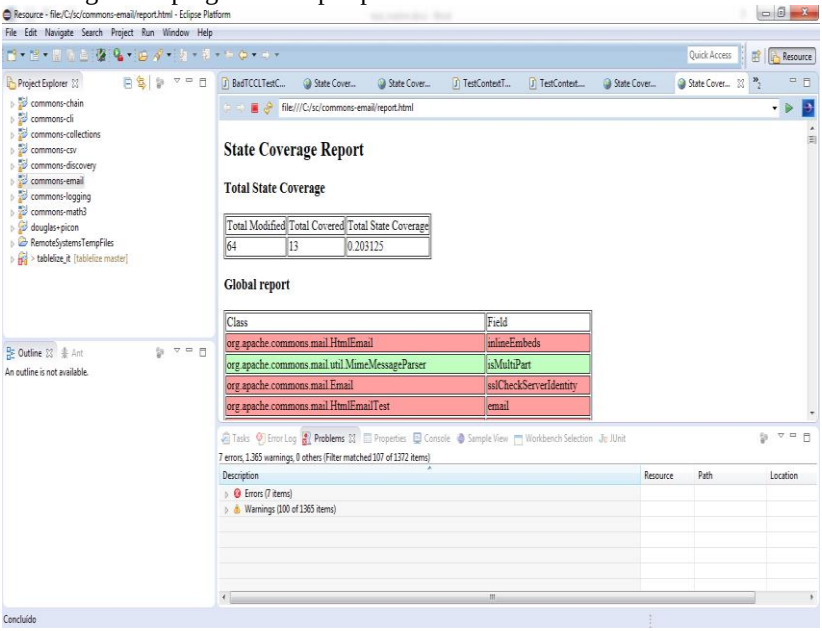
Class	Field
tablelize.Table	tableName
tablelize.Table	rows
tablelize.Fixture	tables
tablelize.Table	positionInFile
tablelize.Table	args
tablelize.Row	row

State Coverage for each test

Test name	Modified States	Covered States	State Coverage
test/FixtureTest.testAsAFixtureIWantToIterateOverTablesWithSameName()V	6	3	0.5

Class	Field
tablelize.Table	tableName
tablelize.Table	rows
tablelize.Fixture	tables

Figura 48: Imagem do plugin do eclipse para o Scova



6 EXPERIMENTOS E RESULTADOS

Para analisar a utilidade do algoritmo e a implementação foram feitos experimentos em 7 projetos de código aberto utilizando a ferramenta Scova. O Quadro 3 lista os projetos escolhidos e seus respectivos endereços web. Todos os projetos analisados estão escritos na linguagem Java e possuem suítes de testes unitários.

Quadro 3: Listagem dos projetos utilizados no experimento

Nome do projeto	**Endereço do projeto**
Tablelize-it	https://github.com/martim00/tablelize_it
Picon	https://code.google.com/p/picon-reflect/
Apache Commons Chain	http://commons.apache.org/proper/commons-chain/
Apache Commons CLI	http://commons.apache.org/proper/commons-cli/
Apache Commons CSV	http://commons.apache.org/proper/commons-csv/
Apache Commons Email	http://commons.apache.org/proper/commons-email/
Apache Commons Discovery	http://commons.apache.org/proper/commons-discovery/

6.1 EXPERIMENTO 1: CORRELAÇÃO ENTRE COBERTURA DE ESTADOS E COBERTURA DE LINHAS DE CÓDIGO

No primeiro experimento buscou-se encontrar uma correlação entre cobertura de estados e cobertura de linhas de código. A análise de correlação tem como objetivo fornecer um número que mostre como duas variáveis variam conjuntamente. Esse número indica a intensidade e a direção dessa relação (LIRA, 2004).

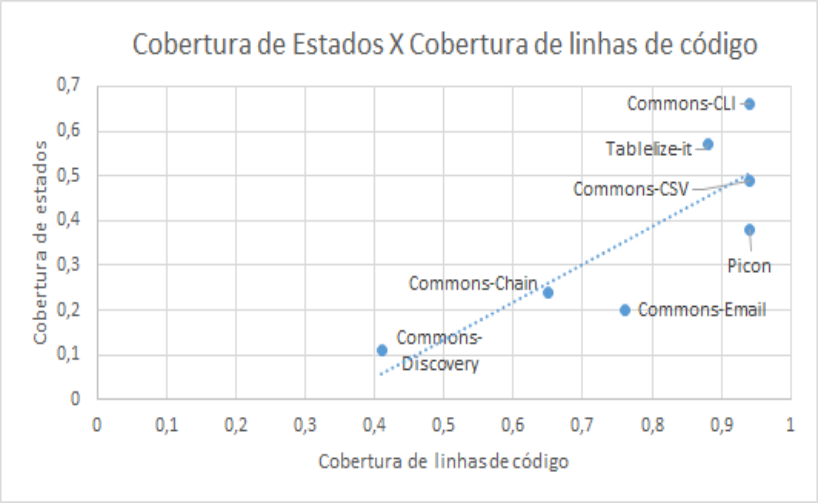
O experimento consistiu inicialmente em medir a taxa de cobertura de linhas de código e a taxa de cobertura de estados para os testes unitários de cada projeto. O Quadro 4 mostra os valores encontrados para as duas métricas.

Quadro 4: Porcentagem de cobertura de estados e linhas de código para os projetos

Projeto	**Cobertura de linhas de código**	**Cobertura de estados**
Tablelize-it	0,88 (88%)	0,66 (66%)
Apache Commons Chain	0,65 (65%)	0,24 (24%)
Apache Commons CLI	0,941 (94%)	0,66 (66%)
Apache Commons CSV	0,94 (94%)	0,49 (49%)
Picon	0,94 (94%)	0,38 (38%)
Apache Commons Email	0,76 (76%)	0,2 (20%)
Apache Commons Discovery	0,41 (41%)	0,11 (11%)

Em seguida plotou-se um gráfico de dispersão para as duas variáveis (Figura 49). Em uma análise visual foi possível identificar uma linha de tendência crescente entre os pontos, acusando uma possível correlação linear positiva.

Figura 49: Gráfico de dispersão para taxas de cobertura dos projetos



Para comprovar esta correlação foi utilizado o cálculo do Coeficiente de Correlação Linear de Pearson para as duas variáveis. Tal coeficiente tem como objetivo identificar de maneira mais precisa o grau de correlação entre duas variáveis. O valor do coeficiente varia em uma escala entre -1 e +1, onde valores mais próximos a -1 demonstra correlação negativa (variáveis são inversamente proporcionais) e valores mais próximos a +1 tem correlação positiva (variáveis são diretamente proporcionais). Valores próximos de 0 (entre -0.3 e +0.3) não demonstram correlação linear (LIRA, 2004).

O coeficiente de Pearson para os dados em análise atingiu um valor de 0,8. Interpretando este coeficiente é possível comprovar uma correlação positiva entre as duas métricas, já que o valor apresentado está próximo de 1.0, o que seria uma correlação positiva perfeita. Uma das conclusões que se pode chegar é que quanto maior a cobertura de estados maior também será a cobertura de linhas de código e vice versa.

É importante ressaltar que este experimento não permite comparações sobre a qualidade das duas métricas, ou seja, qual é melhor ou pior. Apenas faz-se uma exploração das propriedades de ambas com o objetivo de achar alguma correlação entre as duas, o que de fato foi possível.

6.2 EXPERIMENTO 2: ANÁLISE DOS RESULTADOS POR PROJETO

Para o segundo experimento foram escolhidos dois entre os projetos para uma análise mais profunda sobre os resultados. Diferente do primeiro experimento (que não tinha como objetivo comparar as duas métricas) o objetivo neste segundo foi encontrar uma situação real na qual a cobertura de estados apresenta alguma vantagem em relação a cobertura de código.

Os projetos escolhidos foram o *Tablelize-it* e o *Apache Commons Email*.

Para cada um desses projetos foram realizados os seguintes procedimentos:

1. Dentre os atributos não cobertos pelo teste de acordo com a cobertura de estados, buscaram-se aqueles que também estavam adequados quanto à cobertura de código (tinham seus valores executados pelo teste).

2. Para cada um desses atributos foram feitas manipulações no código para que os seus valores ficassem incorretos em algum momento, como se simulassem a inserção de um bug.
3. Ao executar os testes novamente verificou-se que não falharam, ou seja, não evidenciaram o bug inserido.
4. Criou-se um teste com uma asserção verificando o valor do atributo em questão.
5. Ao executar o relatório de cobertura de estados comprovou-se que o atributo agora estava adequado quanto à cobertura de estados

6.2.1 Análise de cobertura de estados para o Apache Commons Email

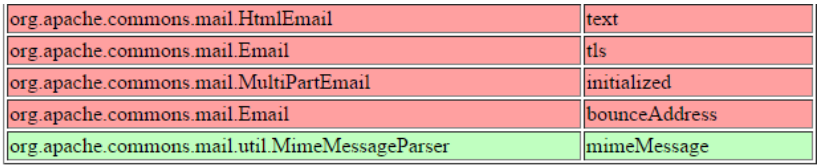
A quantidade de atributos modificados, atributos cobertos e a taxa de cobertura de estados para o projeto Apache Commons Email está listado no quadro 5.

Quadro 5: Relatório de cobertura de estados para o projeto Apache Commons Email

Atributos modificados	Atributos Cobertos	Cobertura de estados
64	13	0,2

A Figura 50 mostra parte do relatório de cobertura de estados apresentado pela ferramenta Scova. O relatório completo pode ser visto no Apêndice A.

Figure 50: Parte do relatório de cobertura de estados apresentado pelo Scova para o projeto Apache Commons Email.



No relatório cada linha representa um atributo da biblioteca Apache Commons Email. A primeira coluna da linha apresenta a classe

do atributo e a segunda coluna apresenta o nome do atributo. As linhas em verde representam os atributos cobertos e em vermelho atributos não cobertos pelo teste.

Como nota-se no relatório (Figura 50), a cobertura de estados acusou que o atributo *bounceAddress* da classe *Email* não estava coberto, ou seja, nenhuma asserção verificava o seu valor. No entanto, ao analisar a cobertura de linhas de código para o projeto em questão notou-se que todas as ocorrências do atributo eram executadas pelo teste. Isso significa uma taxa de 100% de cobertura de linhas de código para esse atributo. Para provar que isso era um problema real foi inserido um bug proposital nesse atributo. O código original e o código modificado com o bug estão listados nas Figuras 51 e 52, respectivamente:

Figura 51: Código original do método setBounceAddress

```
public Email setBounceAddress(final String email)
{
   checkSessionAlreadyInitialized();
   this.bounceAddress = email;
   return this;
}
```

Figura 52: Código modificado com bug do método setBounceAddress

```
public Email setBounceAddress(final String email)
{
   checkSessionAlreadyInitialized();
   this.bounceAddress = "invalid";
   return this;
}
```

Ao executar novamente os testes do projeto notou-se que continuaram passando, sem evidenciar o erro inserido. Isso ratifica a hipótese de que a simples execução de uma linha de código pelo teste não é suficiente para demonstrar a sua validade.

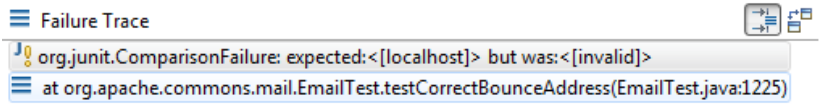
Em seguida foi criado um teste contendo uma asserção no valor desse atributo (Figura 53).

Figura 53: Teste para o atributo *bounceAddress*

```
@Test
public void testCorrectBounceAddress() {
   email.setHostName(strTestMailServer);
   email.setBounceAddress("localhost");
   Session session = email.getMailSession();
   assertEquals("localhost",
           session.getProperty(
                EmailConstants.MAIL_SMTP_FROM));
}
```

Ao executar este novo teste o mesmo falhou, como mostra o console do eclipse (Figura 54), evidenciando o erro inserido.

Figura 54: Console do Eclipse demonstrando o teste *testCorrectBounceAddress* falhando



6.2.2 Análise de cobertura de estados para o projeto Tablelize-it

Foi realizado o mesmo experimento para o projeto Tablelize-it. O Quadro 6 e a Figura 55 apresentam o relatório de cobertura de estados para o projeto e a listagem dos estados cobertos e não cobertos, respectivamente.

Quadro 6: Relatório de cobertura de estados para o projeto Tablelize-it

Atributos modificados	Atributos Cobertos	Cobertura de estados
6	4	0,66

Figura 55: Relatório de cobertura de estados apresentado pelo Scova para o projeto Tablelize-it

Class	Field
tablelize.Table	tableName
tablelize.Table	rows
tablelize.Fixture	tables
tablelize.Table	positionInFile
tablelize.Table	args
tablelize.Row	row

Para este projeto foi escolhido o atributo *positionInFile* da classe *tablelize.Table*. Como não há nenhuma asserção verificando seu valor, ao modificar o método *setPositionInFile* da mesma classe, inserindo um bug proposital os testes do projeto continuaram passando. O código original e o código modificado estão listados nas Figuras 56 e 57, respectivamente. A modificação consistiu no incremento do atributo *positionInFile* em uma unidade.

Figura 56: Código original do método setPositionInFile

```
public void setPositionInFile(int positionInFile) {
   this.positionInFile = positionInFile;
}
```

Figura 57: Código com bug no método setPositionInFile

```
public void setPositionInFile(int positionInFile) {
   this.positionInFile = positionInFile;
   this.positionInFile++;
}
```

Assim como na análise anterior ao verificar o resultado da cobertura de código notou-se que todas as ocorrências de uso do atributo em questão eram realmente executadas pelos testes (como ilustra a Figura 58), ou seja, estavam adequadas quanto ao critério de cobertura de linhas de código.

Figura 58: Relatório de cobertura de linhas de código para a classe Table

```
public class Table {

    // ...

    private int positionInFile = 0;

    public Table(String tableName) {
        this.tableName = tableName;
    }

    public void setPositionInFile(int positionInFile) {
        this.positionInFile = positionInFile;
        this.positionInFile++;
    }

    // ...
```

Ao criar um teste para verificar este atributo (figura 59) comprovou-se que a sua execução falhava por conta do erro inserido, como mostra o relatório de testes do eclipse (Figura 60). No entanto, a diferença é que agora o atributo está adequado quanto à cobertura de estados, como mostra a Figura 61.

Figura 59: Teste cobrindo o atributo positionInFile

```
@Test
public void testPositionInFile() {
   Table table = new Table("");
   table.setPositionInFile(1);
   assertEquals(1, table.getPositionInFile());
}
```

Figura 60: Relatório de testes do eclipse demonstrando o teste testPositionInFile falhando

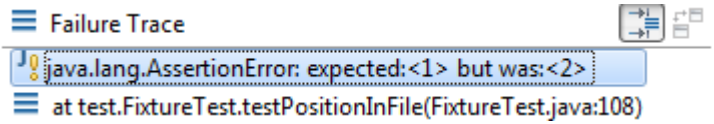


Figura 61: Relatório de cobertura de estados apresentado pelo Scova para o projeto Tablelize-it após o novo teste criado

Class	Field
tablelize.Table	tableName
tablelize.Table	rows
tablelize.Fixture	tables
tablelize.Table	positionInFile
tablelize.Table	args
tablelize.Row	row

É importante ressaltar que este experimento não pretende provar que a cobertura de estados é melhor do que a cobertura de código em todas as situações. O objetivo aqui foi apenas encontrar uma situação real onde o uso da cobertura de estados evidenciaria um problema não capturado pela cobertura de código.

6.3 ANÁLISE DE DESEMPENHO DO SCOVA

Foi realizado ainda um terceiro experimento com o objetivo de verificar o desempenho em tempo de execução da ferramenta Scova. Buscou-se medir o quanto a análise de cobertura de estados acrescenta ao tempo de execução dos testes.

Para cada projeto selecionado no primeiro experimento foram mensurados os tempos de instrumentação e tempos de execução do teste instrumentado e do teste não-instrumentado. Foram coletadas 5 execuções para cada projeto, obtendo-se uma média aritmética destes tempos. Os testes foram executados em uma máquina com um processador de 8 núcleos, 2.10 GHz e 8 Gb de memória RAM. O resultado desta análise está listado no Quadro 7.

Quadro 7: Análise de desempenho completa (instrumentação + execução)

Métrica	Tablelize-it	Commons-email	Commons-csv	Commons-cli	Commons-chain	Commons-discovery	Picon
Execução não instrumentada (ms)	1589	32525	8372	16933	16072	1730	8155
Execução instrumentada (ms)	1677	33726	9645	20538	17671	1802	9999
Instrumentação (ms)	426	8572	1143	915	1739	302	840
Execução + instrumentação (ms)	2103	42298	10788	21453	19410	2014	10839
Tempo acrescentado (ms)	514	9773	2416	4520	3338	372	2684
Total acrescentado (%)	32,34	30,04	28,85	26,69	20,76	21,61	32,91

Como pode-se notar o *overhead* total acrescentado pela instrumentação variou entre 20% e 33% aproximadamente. O *overhead* maior aconteceu no projeto Picon, com 32,91% e o menor aconteceu no projeto Apache Commons Chain, com 20,76%. É interessante notar que a maior parte do tempo acrescentado foi causado pela instrumentação do código. Ao analisar apenas o overhead acrescentado pela execução do teste instrumentado nota-se que esse percentual diminui, como mostra o Quadro 8.

Quadro 8: Análise de desempenho apenas da execução instrumentada

Métrica	Tablelize-it	Commons-email	Commons-csv	Commons-cli	Commons-chain	Commons-discovery	Picon
Execução não instrumentada (ms)	1589	32525	8372	16933	16072	1730	8155
Execução instrumentada (ms)	1677	33726	9645	20538	17671	1802	9999
Tempo acrescentado (ms)	88	1201	1273	3605	1599	72	1844
Total acrescentado (%)	5,53	3,69	15,20	21,28	9,94	4,16	22,61

No Quadro 8 é possível notar que o percentual acrescentado na execução instrumentada fica em torno de 3% e 23% aproximadamente, o que demonstra que o algoritmo de cobertura proposto tem pouco efeito no tempo de execução do teste.

Ambos os resultados são aceitáveis no contexto de análise de critérios de adequação, onde o tempo de execução não é um fator que impacte a sua utilidade. Trabalhos anteriores com cobertura de estados chegaram a apresentar um *overhead* de 70% no tempo de execução (KOSTER, 2008). No entanto, como trabalho futuro espera-se tentar melhorar este tempo de instrumentação.

7 COMPARAÇÃO COM TRABALHOS EXISTENTES

Nas próximas seções são apresentadas as diferenças entre este trabalho e os trabalhos existentes na literatura.

7.1 QUANTO À DEFINIÇÃO DE ESTADOS DE UM PROGRAMA

Como apresentado no Capítulo 2 (Fundamentação Teórica) o presente trabalho utiliza a definição de estados de um programa como proposta por Vanoverberghe et al (2012). Nesta definição o estado de um programa em determinado ponto de sua execução são os valores dos atributos de seus objetos. Diferente da definição proposta por Koster et al (2007) os retornos de métodos públicos aqui não fazem parte do estado. Este trabalho, no entanto, diferentemente de Vanoverberghe et al não faz distinção entre atributos cobertos em diferentes objetos de uma mesma classe (cobertura de estados sensitiva a objetos) por acreditar que esta análise adicional acrescentaria complexidade na leitura dos resultados sem trazer vantagens significativas para a cobertura de estados. No entanto, esta é apenas uma intuição deste autor. Como trabalhos futuros espera-se tentar provar isto.

7.2 QUANTO À DEFINIÇÃO DE COBERTURA DE ESTADOS

O presente trabalho se difere da definição de cobertura de estados proposta por Koster et al (2007) e Vanoverberghe et al (2012) em relação ao elemento de medida para a taxa de cobertura. Assim como na cobertura de código tradicional ambas as definições de cobertura de estados existentes apresentam linhas de código como elemento de medida de cobertura – as linhas de código onde foram feitas as últimas atualizações em um estado do programa (o ODS de Koster e a atualização de estados de Vanoverberghe). Este trabalho entende que uma asserção deve cobrir o estado do programa e não as linhas de código responsáveis pela sua modificação. Por este motivo considera como elemento de medida para a taxa de cobertura de estados os próprios atributos de objetos do programa.

7.3 QUANTO À DISTINÇÃO ENTRE ATRIBUTOS DE TIPO SIMPLES E TIPO COMPOSTO

Outra contribuição deste trabalho em relação aos existentes é a distinção no tratamento de atributos simples e atributos compostos. Nesta proposta um atributo do tipo composto pode ser modificado não só apenas através de atribuições mas também através de chamadas para métodos que modificam sua estrutura. Sem isto muitas modificações em atributos poderiam passar escondidas pela análise de cobertura de estados

7.4 QUANTO ÀS TÉCNICAS UTILIZADAS PARA VERIFICAR SE UM ATRIBUTO ESTÁ COBERTO POR UMA ASSERÇÃO

Saber quando um atributo está coberto por uma asserção é um problema fundamental no cálculo de cobertura de estados. Na prática envolve saber se o mesmo teve influência na asserção. Para resolver isto Koster et al (2007) propõem a utilização de uma técnica conhecida como *program slice*. Essa técnica foi inicialmente proposta por Weiser (1981) como forma de auxílio para a depuração de programas. Consiste em encontrar o subconjunto de instruções de um programa que influenciam no valor de uma variável em determinado ponto da execução. Intuitivamente, é a mesma atividade exercida pelo programador quando tenta encontrar um erro no sistema. Ele foca apenas nos trechos de código que têm relação com a variável sendo depurada, eliminando mentalmente todas as outras instruções. Utilizando esta técnica é possível verificar, a partir da asserção, quais são as instruções que influenciaram no seu resultado. Se o ODS que definiu uma variável pertence a esse subconjunto, então o mesmo é considerado coberto. No entanto, esta é uma análise ineficiente em termos de processamento, pois cada asserção tem seu próprio subconjunto de linhas influentes. Isso exige que a análise de *slice* tenha que ser feita isoladamente para cada asserção (KOSTER, 2008).

Para melhorar isto, Koster (2008) propõe uma outra forma de realizar esta verificação de influência, utilizando *taint analysis*. *Taint analysis* é uma técnica utilizada em segurança computacional para verificar possíveis entradas maliciosas por parte de usuários. Um exemplo é o de um sistema web onde o texto digitado em um formulário acaba sendo executado sem nenhuma verificação no banco de dados. Um usuário malicioso poderia preparar um texto malformado com o objetivo de fazer acessos indevidos no servidor. A técnica consiste em marcar todas essas entradas de usuários com uma *flag*, dizendo que aquele texto

é “contaminado”. A partir daí qualquer outra variável que receba o resultado desse texto passa a ser também tratada como “contaminada”. Para que uma variável seja “descontaminada” a mesma deve passar por uma função que realize tratamentos de segurança (como a remoção de caracteres malformados). Ao chegar na execução da consulta no banco de dados só serão permitidas a execução das variáveis ‘descontaminadas’, lançando uma exceção caso contrário.

Os autores propõem a utilização desta técnica para verificar a influência de um ODS no resultado de uma asserção. Apesar de ter um melhor desempenho do que *program slice*, a técnica ainda adiciona um overhead grande na execução do teste pois para cada instrução do programa é preciso executar uma função de verificação de contaminação para saber se o alvo da atribuição (lado esquerdo) irá ser contaminado pelos identificadores participantes da expressão do lado direito.

Por outro lado, o algoritmo de verificação de influências proposto neste trabalho apresenta uma abordagem “preguiçosa” para a verificação de influências. Essa verificação é feita apenas uma vez para cada asserção do teste. Considere o exemplo da Figura 62.

Figura 62: Exemplo para comparação com taint analysis

```
this.a = 0;
this.b = this.a;
this.c = this.b + this.a;
assertEquals(0, this.c);
```

Na análise de contaminação, a propagação de contaminação deve ser computada para cada instrução no momento em que acontecem. No exemplo, deveriam ser feitas três verificações de contaminação: uma para a atribuição da segunda linha e duas para a atribuição da terceira linha (uma para saber se o atributo b está contaminado e outra para saber se o atributo a está contaminado). Já na análise de influências proposto por este trabalho apenas uma verificação de influência seria necessária: na asserção da linha 4.

Esta diferença de eficiência pôde ser comprovada nos experimentos contidos no Capítulo 6. Enquanto a ferramenta Scova apresentou um overhead de no máximo 33%, a ferramenta proposta por Koster que utiliza *taint analysis* apresentou um overhead de 70%. Segundo o próprio autor metade deste overhead se dá pelo uso de *taint analysis* (KOSTER, 2008).

Vanoverberghe at al (2012), por sua vez, não apresenta nenhum algoritmo para realizar esta verificação de influência em asserções. Os autores se limitam a dizer que este tipo de verificação é feito através de análise de fluxo de informações mas não explicitam como isso é feito em seu trabalho (VANOVERBERGHE et al, 2012).

8 CONCLUSÃO

O presente trabalho apresentou um novo algoritmo para o cálculo de cobertura de estados. Diferente dos trabalhos existentes que utilizam técnicas de outros domínios para saber se a saída do programa é verificada por asserções (KOSTER et al, 2007; KOSTER, 2008) ou nem apresentam como isso é feito (VANOVERBERGUE et al, 2012) este trabalho apresentou um algoritmo próprio de verificação de influências de atributos em métodos baseado na extração de dependências entre identificadores existentes no código.

Por utilizar construções comuns a diversas linguagens orientadas a objetos o algoritmo pode ser facilmente implementado e portado para outras linguagens de programação.

Foi realizada uma implementação do algoritmo na linguagem Java através de instrumentação de código, mostrando resultados promissores tanto em relação à eficiência quanto em relação à praticidade de uso. Esta é a primeira implementação para cobertura de estados que se tem conhecimento disponível para uso e experimentos como código aberto na internet.

Foi possível comprovar a validade do algoritmo proposto e da implementação através de sua aplicação em 7 projetos de código aberto. Os resultados mostraram que existe correlação linear positiva entre cobertura de linhas de código e cobertura de estados.

Em dois desses projetos foram feitas análises mais específicas. Nestes projetos a cobertura de estados foi útil para identificar atributos não verificados por asserções mas que eram executados pelos testes (como comprovado pelo relatório de cobertura de código). Ao inserir erros propositais nesses atributos os testes continuaram passando sem evidenciar o problema. Em uma situação real, estes erros estariam imperceptíveis pelos desenvolvedores.

Este trabalho apresentou ainda uma extensão à definição de estado coberto e modificado para atributos de tipo composto (estrutura de dados). Neste caso, um atributo pode ser considerado modificado se houver modificação na estrutura de seus elementos, diferentemente dos trabalhos existentes que apenas consideram modificado um atributo que é alvo de uma atribuição de valores.

É notável a validade de cobertura de linhas de código como forma de guiar o testador na busca de caminhos não testados e de nenhuma maneira ela pode ser ignorada. No entanto, a cobertura de estados se apresenta como um critério de adequação complementar que vem auxiliar ainda mais o testador na árdua tarefa de validar um programa.

Como trabalhos futuros espera-se melhorar a integração com bibliotecas externas (por exemplo, bibliotecas padrões de linguagens) com o objetivo de identificar de forma mais automatizada possíveis métodos que modificam objetos sem precisar fazer uso de *whitelists*.

Além disso pretende-se incluir outros tipos de construções como estado de um programa, como por exemplo o retorno de métodos públicos e parâmetros passados por referência, como definido em Koster et al (2007) e estender o algoritmo para abranger cobertura de estados sensitiva a objetos como proposta por Vanoverberghe et al (2012). Pretende-se também verificar através de experimentos as vantagens e desvantagens destas abordagens.

Espera-se ainda encontrar um meio mais eficiente de instrumentação de código para diminuir o overhead causado pela execução da análise de cobertura de estados.

REFERÊNCIAS

DIJKSTRA, Edsger W. Chapter I: Notes on structured programming. Academic Press Ltd, 1972.

RAPPS, S.; Weyuker, E.J. Selecting Software Test Data Using Data Flow Information, Software Engineering, IEEE Transactions on, vol.SE-11, no.4, pp.367,375, Abril 1985

GOODENOUGH, J. B. AND GERHART, S. L. Toward a theory of test data selection. IEEE Trans. Softw. Eng. SE-3. Junho de 1975.

MYERS, Glenford J. and SANDLER, Corey. The Art of Software Testing. John Wiley & Sons. 2004.

ZHU, H., HALL, P. A. V., and MAY, J. H. R Software unit test coverage and adequacy. ACM Computing Surveys, 29(4):366–427, 1997.

KOSTER, Ken, KAO David C. State Coverage: A structural Test Adequacy Criterion for Behavior Checking. ESEC/FSE'07, Setembro, 2007.

VANOVERBERGHE, Dries, DE HALLEUX, Jonathan, TILLMANN, Nikolai, and PIESSENS, Frank. State Coverage: Software Validation Metrics beyond Code Coverage. Microsoft Research. Proceedings of the 38th international conference on Current Trends in Theory and Practice of Computer Science. 2012

KOSTER, Ken. A state coverage tool for JUnit. In Companion of the 30th international conference on Software engineering (ICSE Companion '08). ACM. 2008.

ANDREWS, J.H. BRIAND, L.C.; LABICHE, Y., Is mutation an appropriate tool for testing experiments? ICSE 2005. Proceedings. 27th International Conference on Software Engineering, 2005. Maio, 2005

LU, Shan, ZHOU, Pin, LIU, Wei, ZHOU, Yuanyuan, and TORRELAS, Josep. PathExpander: Architectural Support for Increasing the Path Coverage of Dynamic Bug Detection. InProceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture(MICRO 39), 2006.

KITCHENHAM B, CHARTER S. Guidelines for performing Systematic Literature Reviews in Software Engineering No. EBSE 2007-001, 2007.

JabRef Reference Manager Home Page. Disponível em http://jabref.sourceforge.net/. Acesso em: 24 jun. 2013.

BibTex Reference Management Format. Disponível em: http://www.bibtex.org/. Acesso em 23/01/2015.

JUnit framework test. Disponível em: http://junit.org/. Acesso em 26 jan. 2015

MESZAROS, Gerard. xUnit Test Patterns. Pearson Education, Inc./Addison Wesley, 2007.

CHILENSKI, John Joseph. Software Development under DO-178B. Software Engineering. Boeing Commercial Airplanes, Janeiro de 2002.

GOODENOUGH, J. B. AND GERHART, S. L. Toward a theory of test data selection. IEEE Trans. Softw. Eng. SE-3. Junho, 1975.

FRATE, Del, GARG, P.; MATHUR, A.P.; PASQUINI, A. On the correlation between code coverage and software reliability. Software Reliability Engineering Proceedings., Sixth International Symposium on, Outubro 1995

CAI, Xia, LYU, Michael R. The effect of code coverage on fault detection under different testing profiles. SIGSOFT Softw. Eng. Notes 30, 4. Maio, 2005.

STOBIE, Keith. Too Darned Big to Test. Queue Magazine 3, 1, 30-37. Fevereiro, 2005.

INOZEMTSEVA, Laura and HOLMES, Reid. Coverage is not strongly correlated with test suite effectiveness. Proceedings of the 36th International Conference on Software Engineering (ICSE 2014). ACM, Nova York, USA, 2014.

MIRANDA, Breno. A proposal for revisiting coverage testing metrics. Proceedings of the 29th ACM/IEEE international conference on Automated software engineering (ASE '14). ACM, Nova York, USA. 2014.

GAY, Gregory, STAATS, Matt, WHALEN, Michael, HEIMDHAL, Mats. Moving the goalposts: coverage satisfaction is not enough. Proceedings of the 7th International Workshop on Search-Based Software Testing (SBST 2014). ACM, Nova York, USA, 2014.

NAMIN, Akbar Siami e ANDREWS, James. The influence of size and coverage on test suite effectiveness. In Proceedings of the eighteenth international symposium on Software testing and analysis (ISSTA '09). ACM, Nova York, USA, 57-68. 2009.

MEYER, Bertrand, WEI, Yi e ORIOL, Manuel. Is branch coverage a good measure of testing effectiveness?. In Empirical Software Engineering and Verification, Bertrand Meyer and Martin Nordio (Eds.). Springer-Verlag, Berlim, Heidelberg 194-212. 2012.

KANER, C. Fundamental Challenges in Software Testing. Florida Tech, Colloquium Presentation at Butler University, April 2003. Disponível em http://www.kaner.com/pdfs/FundamentalChallenges.pdf. Acesso em: 24 de junho de 2013

MARICK, B. How to misuse code coverage. Proceedings of the 16th International Conference on Testing Computer Software, 1999.

BHANSALI, P. V. The MCDC paradox. SIGSOFT Softw. Eng. Notes 32, 3. Maio, 2007.

WEISER, M. Program slicing. Proceedings of the 5th International Conference on Software, 1981

DUESTERWALD, E, GUPTA, R., SOFFA M. L. Rigorous data flow testing through output influences. Second Irvine Software Symposium, 1992.

HALDAR, V., CHANDRA, D., FRANZ, M. Dynamic taint propagation for java. Proceedings of the 21st Annual Computer Security Applications Conference, 2005.

ZHANG, Tiliang, GAO, Fei, ZHANG, Hua, XU, Yuzong. A Whitelist Detection Mechanism Based on Modsecurity. Proceedings of the 2013 Fifth International Conference on Multimedia Information Networking and Security (MINES '13). IEEE Computer Society, Washington, DC, USA, 34-37. 2013.

SCOVA. Disponível em: < https://github.com/martim00/scova>. Acesso em: 29 jan. 2013.

BRUNETON, Eric, LENGLET, Romain, COUPAYE, Thierry. ASM: a code manipulation tool to implement adaptable systems. France Télécom R&D. Grenoble, França. 2002

LIRA, Sachiko Araki. Análise de correlação: abordagem teórica e de construção dos coeficientes com aplicações. Tese de mestrado. Universidade Federal do Paraná. Curitba, 2004.

TURNER, C.D. e ROBSON, D.J. The State-based Testing of Object-Oriented Programs. Proceedings of the 1993 IEEE Conference on Software Maintenance (CSM- 93), Montreal, Quebec, Canada. Setembro, 1993.

APÊNDICE A – Relatório de cobertura de estados para o projeto Apache Commons Email

Planilha 1: Atributos cobertos para o projeto Apache Commons Email

Atributos cobertos	
Classe	**Atributo**
org.apache.commons.mail.util.MimeMessageParser	isMultiPart
org.apache.commons.mail.EmailAttachment	disposition
org.apache.commons.mail.DefaultAuthenticator	authentication
org.apache.commons.mail.util.MimeMessageParser	htmlContent
org.apache.commons.mail.EmailAttachment	name
org.apache.commons.mail.util.MimeMessageParser	attachmentList
org.apache.commons.mail.resolver.DataSourceCompositeResolver	dataSourceResolvers
org.apache.commons.mail.AbstractEmailTest	fakeMailServer
org.apache.commons.mail.MultiPartEmailTest	email
org.apache.commons.mail.EmailAttachment	description
org.apache.commons.mail.util.MimeMessageParser	plainContent
org.apache.commons.mail.EmailAttachment	path
org.apache.commons.mail.util.MimeMessageParser	mimeMessage

Planilha 2: Atributos cobertos para o projeto Apache Commons Email

Atributos não cobertos	
Classe	Atributo
org.apache.commons.mail.HtmlEmail	inlineEmbeds
org.apache.commons.mail.Email	sslCheckServerIdentity
org.apache.commons.mail.HtmlEmailTest	email
org.apache.commons.mail.Email	toList
org.apache.commons.mail.Email	emailBody
org.apache.commons.mail.Email	sslSmtpPort
org.apache.commons.mail.Email	hostName
org.apache.commons.mail.MultiPartEmail	primaryBodyPart

org.apache.commons.mail.Email	socketConnectionTimeout
org.apache.commons.mail.MultiPartEmail	boolHasAttachments
org.apache.commons.mail.resolver.DataSourceBaseResolver	lenient
org.apache.commons.mail.Email	bccList
org.apache.commons.mail.EmailAttachment	url
org.apache.commons.mail.Email	debug
org.apache.commons.mail.Email	authenticator
org.apache.commons.mail.MultiPartEmail	container
org.apache.commons.mail.EmailTest	email
org.apache.commons.mail.HtmlEmail	html
org.apache.commons.mail.Email	headers
org.apache.commons.mail.Email	ssl
org.apache.commons.mail.Email	session
org.apache.commons.mail.Email	popHost
org.apache.commons.mail.HtmlEmail$InlineImage	mbp
org.apache.commons.mail.Email	startTlsEnabled
org.apache.commons.mail.ImageHtmlEmail	dataSourceResolver
org.apache.commons.mail.Email	replyList
org.apache.commons.mail.Email	popBeforeSmtp
org.apache.commons.mail.Email	message
org.apache.commons.mail.Email	startTlsRequired
org.apache.commons.mail.Email	charset
org.apache.commons.mail.Email	socketTimeout
org.apache.commons.mail.Email	contentType
org.apache.commons.mail.Email	subject
org.apache.commons.mail.Email	ccList
org.apache.commons.mail.Email	smtpPort
org.apache.commons.mail.ImageHtmlEmailTest	email
org.apache.commons.mail.MultiPartEmail	subType
org.apache.commons.mail.Email	fromAddress

org.apache.commons.mail.resolver.DataSourceClassPathResolver	classPathBase
org.apache.commons.mail.Email	sentDate
org.apache.commons.mail.resolver.DataSourceUrlResolver	baseUrl
org.apache.commons.mail.HtmlEmail$InlineImage	cid
org.apache.commons.mail.Email	popPassword
org.apache.commons.mail.Email	sslOnConnect
org.apache.commons.mail.Email	content
org.apache.commons.mail.HtmlEmail$InlineImage	dataSource
org.apache.commons.mail.Email	popUsername
org.apache.commons.mail.HtmlEmail	text
org.apache.commons.mail.Email	tls
org.apache.commons.mail.MultiPartEmail	initialized
org.apache.commons.mail.Email	bounceAddress

Fonte: desenvolvido pelo autor

APÊNDICE B – Publicações

Publicação 1:

Título

Um algoritmo para cobertura de estados

Conferência

XVII Iberoamerican Conference on Software Engineering - CibSE

Qualis

B4

Resumo

Cobertura de estados é um critério de adequação de testes de software que mede a quantidade de estados modificados durante a execução de um teste que foram cobertos através de asserções. O presente trabalho propõe um algoritmo para o cálculo de cobertura de estados baseado na instrumentação de construções comuns a linguagens orientadas a objetos, como atribuições, retorno de métodos e chamadas de funções. O algoritmo identifica a influência de um atributo no resultado de uma asserção através de um novo cálculo de influências de variáveis em métodos. O artigo apresenta ainda uma implementação através da instrumentação de bytecode Java e experimentos utilizando essa ferramenta em um projeto de código aberto.